Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

Antti Laisi

A reference architecture for event-driven microservice systems in the public cloud

Master's Thesis
Halkia, November 25, 2019

Supervisor: Professor Mario Di Francesco
Advisor: Erkki Pulliainen M.Sc. (Tech.)

Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

ABSTRACT OF MASTER'S THESIS

Author: Antti Laisi

Title:
A reference architecture for event-driven microservice systems in the public cloud

Date:	November 25, 2019	**Pages:**	72
Major:	Computer Science	**Code:**	SCI3042
Supervisor:	Professor Mario Di Francesco		
Advisor:	Erkki Pulliainen M.Sc. (Tech.)		

The emergence of public cloud computing platforms has had a profound effect on how software is being developed. To take advantage of many of the features of cloud platforms, software architecture of applications must aligned with the characteristics of cloud services. Where systems designed for traditional data center deployments have typically consisted of a single large application and a centralized data store, systems targeting cloud platform have become distributed applications.

The microservice architecture is a software architecture style for building distributed systems that consist of autonomous services, each responsible for a single problem domain. Decomposing an application to individual components makes is possible to utilize cloud platform features such as scaling each part of the system according to load and performance.

Enterprise applications are the context where the microservice architecture pattern is typically applied. These applications are large, long-lived, in state of constant change and highly integrated to other systems. But building complex enterprise applications as distributed systems poses architectural challenges on how to build a system that is evolvable, maintainable and understandable.

This thesis describes patterns for building microservice systems that can scale to a large amount of services while retaining the autonomy the services and the maintainability of the system as a whole. A key factor in these patterns is the use of events for communication between the different components of the system. The thesis then presents a reference architecture on how such a system can be developed by utilizing managed services of a public cloud platform.

Keywords:	Software architecture, microservice architecture, distributed systems, cloud computing
Language:	English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tieto-, tietoliikenne- ja informaatiotekniikan maisteriohjelma

DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Antti Laisi		
Työn nimi:	Viitearkkitehtuuri tapahtumapohjaiselle mikropalveluarkkitehtuurille pilvipalveluissa		
Päiväys:	25. marraskuuta 2019	**Sivumäärä:**	72
Pääaine:	Tietotekniikka	**Koodi:**	SCI3042
Valvoja:	Professori Mario Di Francesco		
Ohjaaja:	Diplomi-insinööri Erkki Pulliainen		

Lisääntyvä pilvipalveluiden käyttö on vaikuttanut merkittävästi siihen, millaisia sovelluksia kehitetään. Sovelluksen arkkitehtuurin täytyy olla suunniteltu siten, että pilvipalveluiden ominaisuuksia voidaan hyödyntää. Sovellukset, jotka ovat suunniteltu ennen pilvipohjaisia arkkitehtuureja koostuvat tyypillisesti yhdestä suuresta asennettavasta komponentista ja keskitetystä tietovarastosta. Pilvipalveluiden myötä tämän mallin sijaan on ruvettu rakentamaan hajautettuja järjestelmiä.

Mikropalveluarkkitehtuuri on ohjelmistoarkkitehtuuri, jossa hajautettu järjestelmä koostetaan yksittäisistä erillisistä palveluista. Jokainen palvelu vastaa järjestelmän tietystä toiminnosta tai osa-alueesta. Arkkitehtuuri, jossa sovellus on pilkottu pieniin autonomisiin komponentteihin mahdollistaa monien pilvipalveluiden ominaisuuksien (kuten kuorman mukaisen skaalauksen) käytön.

Monimutkaiset yritysjärjestelmät ovat kenttä, jossa mikropalveluarkkitehtuuria tyypillisesti käytetään. Nämä järjestelmät ovat suuria, jatkuvan muutoksen alaisia ja moninaisin tavoin integroituneita useisiin muihin järjestelmiin. Monimutkaisten yritysjärjestelmien rakentaminen mikropalveluarkkitehtuurilla luo omat haasteensa siinä, miten järjestelmästä saadaan ylläpidettävä, jatkokehityskelpoinen ja ymmärrettävä.

Tämä diplomityö kuvaa malleja mikropalvelujärjestelmien rakentamiseen siten, että järjestelmän kasvaessa yksittäiset mikropalvelut pysyvät erillisinä ja autonomisina sekä järjestelmä kokonaisuutena pystyy ylläpidettävänä. Avainrakenne näiden tavoitteiden saavuttamisessa on tapahtumien käyttö tiedon välittämisessä palveluiden välillä. Diplomityössä esitetään sitten viitearkkitehtuuri miten nämä mallit voidaan ottaa käyttöön julkisessa pilvipalvelussa.

Asiasanat:	Ohjelmistoarkkitehtuuri, mikropalveluarkkitehtuuri, hajautetut järjestelmät, pilvipalvelut
Kieli:	Englanti

Acknowledgements

This thesis is dedicated to my wife Jaana, who in the acknowledgements of her PhD dissertation paper promised me that she would learn how to cook — I now promise to learn to do the dishes.

Halkia, November 25, 2019

Antti Laisi

Abbreviations and Acronyms

ALB	Application Load Balancer
API	Application programming interface
AWS	Amazon Web Services
BFF	Backend for frontend
BPEL	Business process execution language
CAP	Consistency, availability and partition tolerance
CRUD	Create, read, update, delete
deb	Debian package
ECS	Amazon Elastic Container Service
ECR	Amazon Elastic Container Registry
FaaS	Function as a service
HTTP	Hypertext transfer protocol
IaaS	Infrastructure as a service
IPC	Inter-procedure call
JSON	JavaScript object notation
MOM	Message oriented middleware
MSK	Amazon Managed Streaming for Kafka
PaaS	Platform as a service
RDS	Relational Database Service
REST	Representational state transfer
RFC	Request for comments
RPC	Remote procedure call
RPM	RPM package manager
SaaS	Software as a service
SNS	Amazon Simple Notification Service
SQS	Amazon Simple Queue Service
URL	Uniform resource locator

Contents

Chapter 1

Introduction

The development and adoption of **cloud computing** has changed how companies are deploying software. While previously applications were designed to be run in private data centers and deployed to a small number of physical servers or virtual machines, now systems are being developed for public cloud platforms.

In the first wave of adopting cloud computing, applications built with existing software architecture structures were simply migrated to the cloud [22]. However, software architecture patterns designed for small-scale data center deployments do not necessarily allow taking full advantage of running application in the cloud. Unlike previous data center environments, cloud computing platforms are elastic [22]. In an elastic environment, a system can react to changing workloads by automatically scaling the number of used cloud platform resources [22]. Making use of the opportunities offered by these new deployment environments required finding new ways of building software.

The microservice architecture is one solution to the challenge of finding ways of building applications that enable efficient use of cloud computing features such as scaling, on-demand capacity and platform as a service. The emergence of the microservice architecture rises from the tradition of large enterprise applications and from the challenges faced in keeping such systems maintainable, scalable and evolvable [35].

In the microservice architecture, an application is decomposed into small independent components called microservices [40]. These services are separate, isolated processes that communicate over lightweight protocols such as HTTP and messaging [15]. Each service is responsible for a certain business feature of the application and services can be independently scaled, deployed or upgraded [31]. Complex business processes can be executed by composing workflows with multiple participating services [32]. In the microservice

architecture, the decomposed microservices together form the aggregated application, the **microservice system** [5].

1.1 Problem statement

Although considerable research has been devoted to defining microservice architecture and especially to what exactly passes for a microservice, rather less attention has been paid to how to compose a system from the decomposed services. Applications built with microservice architecture are by definition distributed systems, and while each microservice in itself might be a simple component, the distributed system as whole becomes inevitably complex [35]. In adopting microservice architecture, there is a risk of ending up with a distributed system with components tightly interconnected on multiple levels – a distributed monolith, a system with all the downsides of both distributed and monolithic applications [35]. Successful application of microservice architecture requires a solution where the level of connectedness, coupling, is kept in check and each component has a single purpose, in other words the component is cohesive [29].

The purpose of this thesis is to present a **reference architecture** for building the composition, the microservice system, and describe how such a system can be implemented in the public cloud. The reference architecture consists of patterns for building microservices that lead to low coupling between the components in the distributed system but enable composing complex and stateful business processes spanning multiple cohesive services. Due its prevalence in cloud-based services, **Amazon Web Services** is chosen as the use case for a public cloud platform in this thesis.

In particular, the scope of the thesis is to describe and evaluate software architecture structures that enable building architecturally scalable microservice systems from decomposed individual components. In the resulting microservice system, each microservice should be able to be evolved, deployed and scaled independently. The system as a whole should be maintainable and it should be able to take advantage of cloud computing platform features such as elasticity and platform as a service capabilities.

The research questions for the thesis are:

- **RQ1**: How to manage state in a loosely coupled microservice system?
- **RQ2**: How to compose complex business processes from individual microservices?

- **RQ3**: How an event stream backbone for a microservice system can be built with Amazon Web Services?
- **RQ4**: How coordinated processes composed from individual microservices can be implemented in Amazon Web Services?

1.2 Methods

The study is conducted as a narrative literacy review of academic publications and industry literature. The thesis first forms an architectural model of an event-based microservice system based on the literature. The model consists of a collection of patterns for building loosely coupled microservice systems. Then, as a case study, the architectural model is applied to the Amazon Web Services public cloud platform. The output of the case study is a reference architecture for building event-based microservice systems on the Amazon public cloud.

1.3 Structure of the thesis

The rest of the thesis is organized as follows. Chapter 2 describes the context of enterprise applications, details monolithic and microservice architectures and discusses pitfalls of distributed systems. Chapter 3 builds up step by step a reference architecture for a loosely coupled microservice system. Chapter 4 presents how patterns in the reference architecture can be implemented using Amazon Web Services. Chapter 5 evaluates the reference architecture and cloud implementation model and presents discussion on the relations between microservice architecture, enterprise applications and cloud computing. The discussion also includes empirical observations of the author from building microservice systems in the industry. Finally, Chapter 6 concludes the thesis and describes possibilities for future research.

Chapter 2

Enterprise application architecture

The microservice architecture is a strategy for breaking software down into isolated components that then form a distributed system [4]. This is a complex with a large overhead and its own downsides [28]. Therefore, the microservice architecture typically is a valid choice only in the cases where the system in bound to become complex in any case — in the case of **enterprise applications**.

Enterprise application is a general term for a software system that has some of these characteristics: the application is integrated to several other application, has a large codebase, processes and persists large volumes of data and is used by a concurrent users [13]. Most of all, enterprise applications are in the state of constant change as the business requirements and priorities change to match needs of the enterprise [13].

The rest of this chapter describes software architecture styles for building enterprise applications and defines the concept of a distributed system.

2.1 Monolithic architecture

An enterprise application has been traditionally built and deployed as a single unit [15]. In the context of the microservice architecture, this kind of application is called a **monolith**. A software architecture can be characterized as monolithic if its components cannot be executed independently [12]. Any change in the monolithic system requires rebuilding and redeploying the entire application [15].

A monolithic application typically includes the user interface (for example, HTML and Javascript run in a web browser), server-side application

code (e.g., Java) for a range of features and database used for persistent data storage [15]. The applications processes user requests by running code that realizes the **business logic** for a feature and uses a database to store and retrieve data [15]. Figure 2.1 presents an example of a monolithic applicatio.

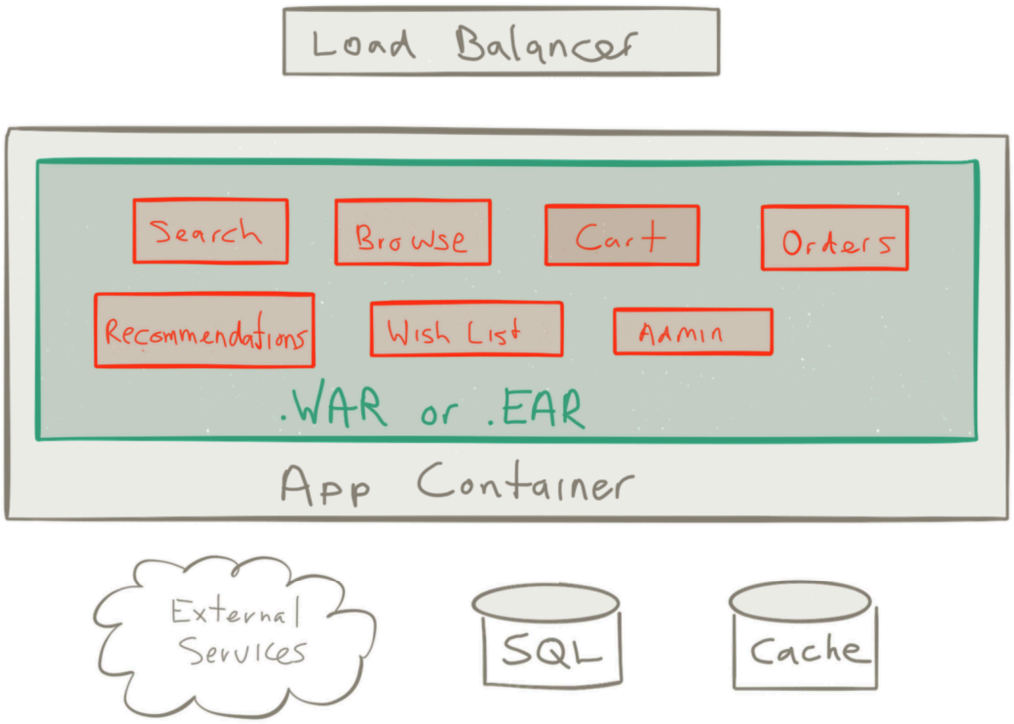

Figure 2.1: A monolithic enterprise application built with Java [4].

Within a monolithic enterprise application, the architecture is typically layered into the presentation layer consisting of the user interface (and possible other external entrypoints), the service layer responsible for the business logic and the data access layer [24]. As shown in figure 2.2, the layered architecture is fairly simple conceptual model that is easily communicated to developers.

However, as enterprise systems have long lifespans are are under constant change, the business logic tends to spead over to all the layers over time, all the way from the user interface to the data stores [37]. The initial separation of duties between components tends to erode and the code becomes deeply interconnected and complex [37]. Constant refactoring is required to prevent this technical debt [37]. The pace of changes can then lead to a growing technical debt as quick fixes and kludges are added to cope with ever-changing requirements [37]. Over time, the development of the system grinds to a halt

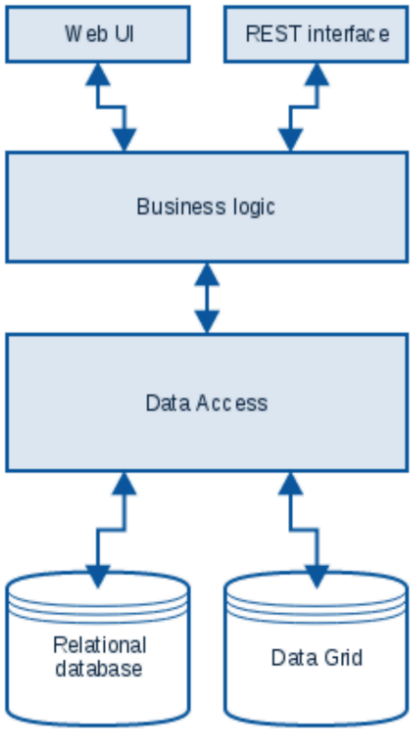

Figure 2.2: Layered architecture for a monolithic enterprise application [24].

and an extremely painful rewrite is started. The microservice architecture tries to tackle this pain point of the enterprise application lifecycle, to create a system that is composed of individual components that can be reused and replaced instead of rewriting large applications [32].

Monolithic applications also have properties that prevent taking advantage from the elasticity intrinsic to cloud computing environments. Scaling a monolithic application is performed by duplicating the monolithic application on multiple servers [15]. This presents some challenges: while only a single subset of the application might form the performance bottleneck, one is always forced to replicate the whole deployment unit [44]. This leads to inefficient use of resources and increased infrastructure costs [44]. Furthermore, as the deployment unit of a monolith is the entire application, a failed deployment can take down the whole system [44]. The single point of failure in the deployment pipeline does not encourage for the use rapid automated deployments possible in cloud environments.

To build large enterprise applications that can better take use cloud plat-

form capabilities, remain maintainable and allow for reacting to changes in the business and developing new features, one option is to leave behind the monolith and step into the domain of distributed computing with the microservice architecture.

2.2 Microservice architecture

In the microservice architecture, the application is split into modular services where the boundary of the module is an operating system process [37]. This is a stark contrast to monolithic applications where module boundaries are defined in terms of programming language features [37]. This microservice approach of divide and conquer is about applying the principles of Unix design to building enterprise applications [9].

The Unix philosophy states three guidelines for application design [39]:

- Write programs that do one thing and do it well;
- Write programs to work together;
- Write programs to handle text streams, because that is a universal interface.

Instead of a single monolithic application, a microservice system is composed of autonomous services that "do one thing and do it well" [27]. These services expose interfaces to other services by using a universal technology-agnostic protocols (e.g., JavaScript object notation, JSON) and complex interactions can be performed by collaborating microservices [27]. The principle of the microservice architecture is indeed to follow the Unix architectural guidelines to achieve a modularized, reusable component architecture [9].

A component in the microservice architecture is a single microservice. A service is built around a single business capacity and it is an autonomous deployment unit that can be individually deployed, scaled (as in figure 2.3) or upgraded [15]. As a single microservice is an autonomous deployment unit, it can be developed using the programming language best suited for the responsibilities of that specific service [27].

The question then quickly becomes how to to define this unit. Despite the name, the size of a microservice is not constrained in lines of code but in the cohesiveness of its responsibilities [35]. If the service is cohesive i.e., it has a single, clearly defined purpose business purpose its exact size in lines or megabytes does not matter.

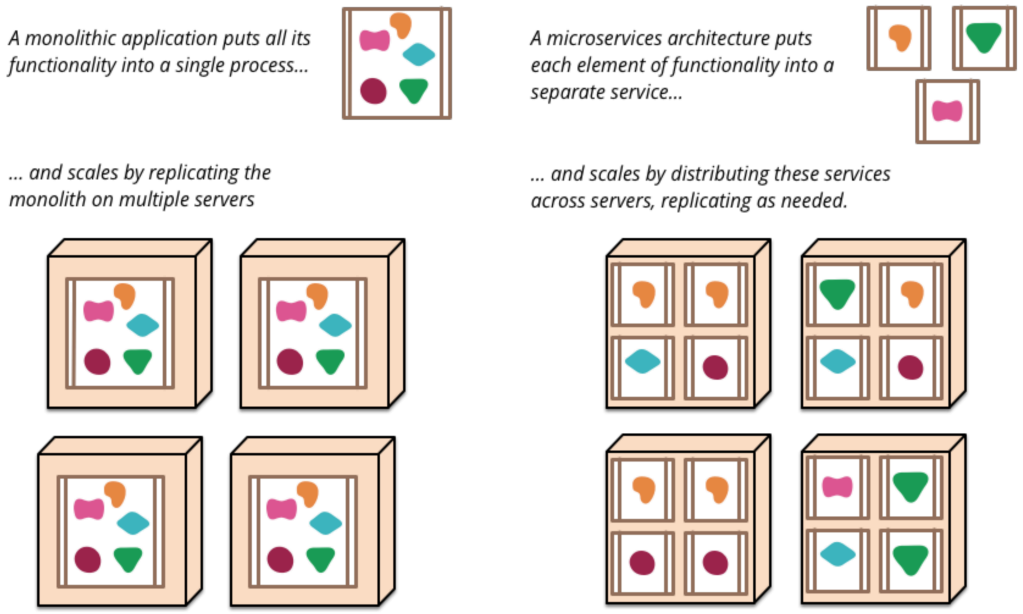

Figure 2.3: Scaling a monolith vs. microservices [15].

The individual microservices together form the aggregate application called the microservice system. In this system, complex business processes are performed with operations spanning multiple microservices. To achieve this, each microservice needs to be reachable by an application programming interface (API) exposing the operations of the service to the microservice system [35]. Beyond these APIs, each microservice is opaque to the other services [35].

The API techologies used with microservices fall into two categories: synchronous and asynchronous [37]. In the former, the caller of the microservice API is expecting a response. An example of a synchronous API typically used with microservices is a RESTful HTTP interface [15], where HTTP methods and URLs are used to define API operations. In contrast, in an asynchronous API, the service will act on a call at some point in the future. A microservice consuming messages from a messaging system queue (such as RabbitMQ) is an example of an asynchronous API [35]. As the APIs are the only entry points to a microservice, designing the interfaces is a key step in building a microservice system.

2.3 Distributed systems

Applications built with the microservice architecture are by definition distributed systems [37]. In a monolithic application, communication between different parts of the application is performed with function calls inside the same operating system process. However, the communication takes place using API calls over the network in a decoupled microservice system [21]. This distinction immediately forces the microservice implementation to deal with aspects such as timeouts, latencies and failures which then raises the complexity of the implementation [44].

Deutsch and Gosling [38] give a thorough overview of the challenges posed by the paradigm shift from a single process monolithic application to a distributed microservice system. In a microservice system, remote integrations have to take into account the latency in calls from one service to another. Synchronous integrations (such as HTTP calls) combined with long interaction chains of services calling services lead to high latencies, even when all services are functioning properly. As every integration has latency, chatty fine-grained interfaces cause high latencies in remote calls [38]. When failures start occurring in the services or the network becomes unreliable or saturated, errors start cascading from one service to another [4]. Network-based APIs also have very different security aspects than local function calls [38]. Some frameworks attempt to blur the distinction between a local and remote call by presenting programming language interfaces to remote APIs that appear as function calls [43]. This is typical of the remote procedure call (RPC) paradigm [43]. Approaches such as these, where the distributed nature of interactions is hidden from the programmer, make the integrations between components in the system fragile and prone to unintended behavior in the case of failures [8].

One issue that is peculiar to distributed systems as opposed to monolithic applications is the problem of reaching consensus. One microservice calling an API of another service can only initiate the interaction, but if the interaction fails (e.g. due to network failure) the service has no knowledge on whether the operation was completed or not. Complex acknowledge systems can then be built alleviate this uncertainty, but no chain of *acks* and *nacks* can guarantee that the parties have consensus on the outcome of the interaction [23]. This is known as the Byzantine Generals problem [23]. Like a general sending a messenger over no man's land, in distributed computing calling a remote integration should be considered a best-effort action and measures should be taken in the case that the message never arrives in its intended destination. Planning for failure is a crucial part of distributed

system design and of applying the microservice architecture [37].

The distributed nature of microservice systems is a double-edged sword. The system can be more scalable if structured correctly [37]. But this is at the cost of complexity. Understanding, designing and debugging distributed systems requires significantly more effort and more skilled developers than monolithic systems [37].

Chapter 3

Event-driven microservice systems

3.1 A microservice system

By definition, an application built with the microservice architecture can not consist of a single microservice – this would be a monolithic application. Microservices always come in **systems** [5]. And it is in the design of the system, and not of a single microservice, where all the challenges of microservice architecture are faced. A microservice system consisting of dozens or hundreds of services can be seen as graph with dozens or hundreds of vertexes and hundreds or thousands of edges (see figure 3.1). A distributed system architecture at this scale can deteriorate into a distributed big ball of mud if clear architectural principles on service interaction are not set up and followed.

In the Art of Unix programming [34], Eric S. Raymond states: *"The only way to write complex software that won't fall on its face is to hold its global complexity down — to build it out of simple parts connected by well-defined interfaces, so that most problems are local and you can have some hope of upgrading a part without breaking the whole."* This very well describes the architectural design that a microservice system should be striving for. Each microservice should be **simple**: a cohesive component with clear responsibilities and documented public interface. The microservice system as a whole is then built as a composition of these components.

But for what should a single microservice be responsible for and how should one model the public interface of a a service? This chapter contains a collection of patterns that offer some answers to these question. Starting with what is at the core of almost any system — data.

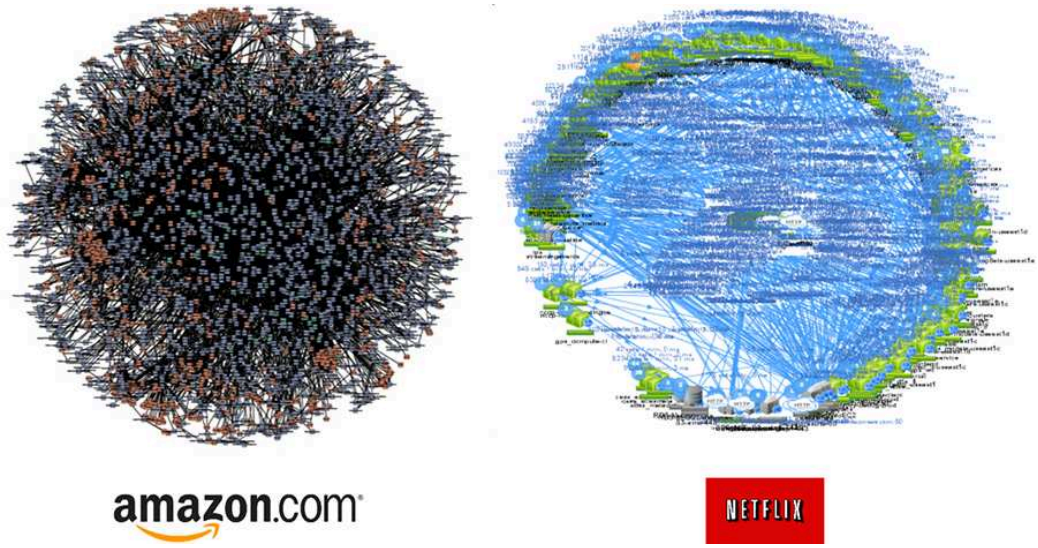

Figure 3.1: "Deathstar diagrams" visualizing connections between microservices in Amazon and Netflix microservice systems.[1]

3.2 State, an elephant in the room

State is defined here as all the data in the entire system. This includes persistent data, such as customer records saved to databases, and transient data, such as populated caches and in-memory state of unfinished business processes.

A buzzword for software architecture designs, frameworks and libraries has for several years now been **stateless**. A stateless solution is pictured to offer "web scale" performance and/or scalability. Often there are truth behind these claims if we examine only a single library of framework. However, if we examine the system as a whole, these stateless solution usually mean that the **state** is just pushed somewhere else, often either to a persistent data store or to the client device.

One of the hardest challenges in large distributed systems is finding the optimal (of even feasible) system design for storing state. A monolithic application typically stores all persistent data in a single database [15]. This enables the application to always have a consistent and up-to-date view on the data where all application components can read and modify the data. However, as visualized in figure 3.2, this approach does not translate well

[1]Image source:
`https://www.appcentrica.com/the-rise-of-microservices/`

to building microservice systems. One service writing data to a data store for another service to read creates tight coupling between those services and makes changes both to the services and to the data format (e.g., in the relational database schema) painful and costly [29]. This IPC-over-database is an antipattern that leads to a system with the worst sides of both monolithic applications and microservice systems [4].

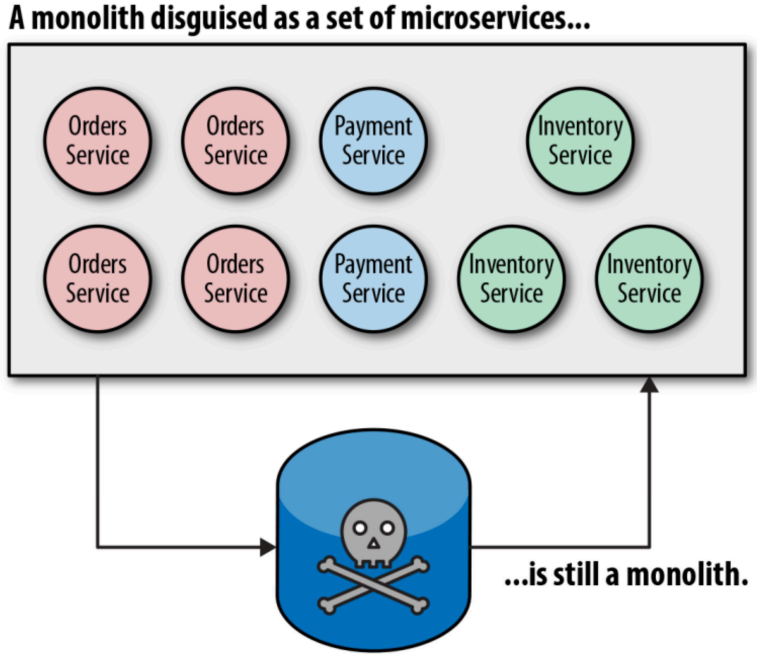

Figure 3.2: Antipattern of microservices communicating over stored database state [5].

An architecturally scalable pattern for storing persistent data in a microservice system is to use **a separate private data store for each service** (see figure 3.3) [35]. When designing the system structure, one should strive for a design where each data entity (e.g. `Customer` or `Account`) should have one and only one owning service which is the source of truth for that piece of data [32]. Other services can store copies of attributes related to the entity for reading purposes, but all modifications must take place through the service that owns the entity [32].

However, using a private data store for each service leads to a new set of problems not typically present in monolithic applications. Data is scattered in dozens of data stores and modifications and reads must somehow be

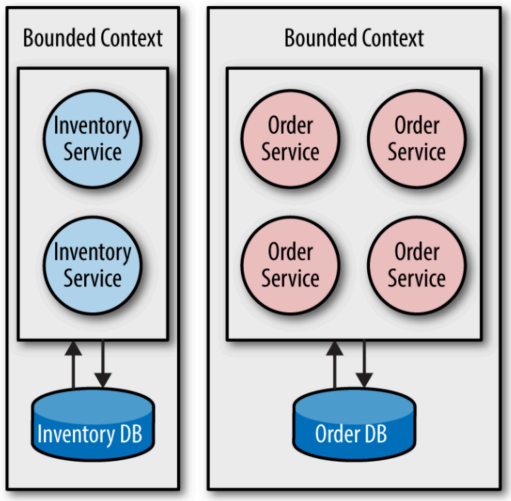

Figure 3.3: Two microservices each with their own data stores. The bounded context is also the scope of strong consistency [5].

coordinated and synced.

3.2.1 Data consistency

If each microservice in the system has a private data store, how can the system as a whole be kept in consistent state where all data is always up-to-date and not conflicting? The answer simply is: **it can not**.

In distributed computing, in general and especially in a microservice system, data consistency is a luxury that is only available locally and never globally. A data store for a single microservice can very well be consistent, however the service can make no assumptions about the state of the data stores belonging to other services. As noted by Helland, each service has a separate clock and a separate concept of *now*, but there is no global now, no global consistency in the system [17]. All interactions between the services must pass trough *then*, the zone between the services where a consistent now is unknown [17]. The systems as a whole is then in a constant state of inconsistency. As an example, a service can hold a `CustomerAddress` record in a private data store that is related to an entity already deleted by the `Customer` service, and this service will be only be notified of the deletion sometime in the future.

This scattering of data storage has also some advantages. Using a private data store for each services enables one to choose the best-fit technology

that fits the requirements if that service. If local strong consistency guarantees are required, the data can be stored in a ACID-compliant relational database with a high isolation level. If local consistency can be sacrifed for scalability or performance, an eventually consistent NoSQL data store can be used. Also, the pattern enables the use of more purpose-specific data store solutions such as search indexes (e.g., Apache Solr or Elasticsearch) or today's equivalent for the humble flat file: Amazon Simple Storage Service (S3), perhaps in combination with techniques such as optimistic locking [13] to achieve some level of local consistency.

3.2.2 Distributed transactions

Historically, **distributed transactions** have been one option for expanding the scope of consistency to cover multiple data stores. In a distributed transaction, a transactions manager calls each participant (e.g. a service with a SOAP over HTTP API) in order to first prepare and lock the changes to be made and if this succeeds, each participant is called to commit the changes [41]. Theoretically, the data modification as a whole is atomic – after the transactions completes, either all changes have been persisted in all data stores or all changes have been reverted.

So one might be tempted to use transactions covering multiple services to atomically update several data stores spanning multiple services. Distributed transactions however are a source of performance problems as data stores are all holding locks while remote procedure calls are being performed by the transaction manager. It is also not possible to use distributed transactions to coordinate long lived operations, for example a use case where a manual approval by a user is required. In a microservice system, using distributed transactions only works when all microservices participating the the transaction are available, so the availability of the coordinated operation is only the product of availabilities of all the participating services.

Distributed transactions are also typically not supported by some newer data stores and messaging or streaming products, which puts constrains on the technology choices available in the system [35].

But most important of all, the two-phase commit protocol simply **reduces** the probability of errors and gives a false sense of security for the programmer. This is a classical example of the Byzantine generals problem. The two-phase commit protocol is only sending an additional messenger over no man's land, but that additional message from the general can be lost just as easily as the preceding messages. If a participant in a two-phase commit transaction crashes (or a network failure occurs) in the commit phase, the state of the transaction as a whole is undefined [41]. If the programmer re-

lied on distributed transactions to prevent this exact problem, the system is not prepared to cope with this situation, and the conflict must somehow be resolved manually. If one is designing a busy system with potentially dozens or even hundreds of data stores, manual conflict resolution is not an option.

But if distributed transactions are out of the picture, what pattern should one use in a microservice system to coordinate state modification between the services?

3.3 Events as a communication mechanism

An **event** is defined in this context as a published fact of something that has already occurred. This fact describes a notable thing that has happened in either inside the system of outside of it [26]. In the **event-driven architecture**, events are sent for notable occurrences and components perform business tasks by acting on the events [26]. The event-driven architecture is loosely coupled as the the component sending the event has no knowledge on the recipients or how or when the event will be processed [26].

In an event-based microservice system, the default communication channel between different microservices should be publishing and consuming events. Whenever a microservice changes the state of its domain, either by updating a data store, or calling an external integration, the service should **publish** an event. The published event should carry the full state of the changed domain entity. Services can **subscribe** to the types of events they are interested in, and a published event is then **consumed** by all subscribers (see figure 3.4).

A trigger for publishing an event can be the input from a user, for example, a user modifying their contact address in a mobile application. The new contact information is received by a `Customer information service` which updates the data in the services private data store, and if this is successful, the service publishes a `CustomerInformationChanged` event. The event does not contain only the identity (primary key) of the `CustomerInformation` entity, but the whole data set with both changed and unchanged attributes. Other microservices that are subscribed to receive `CustomerInformationChanged` events then consume this event and update their own data stores or start business processes such as sending the user a personalized marketing message.

For event-driven microservice systems to support new features or changes in the future, the publisher of the event does not know the consumer(s) of the event [37]. A microservice simply blindly broadcasts all changes that happen in the services own private state and domain, and one or more service might or might not be interested in these events. So in the customer

information example above, let us say that due to regulatory changes, a fraud prevention feature is set up next year. The feature introduces a requirement that all changes for customer postal addresses must be verified against a central registry to prevent identity theft. Since the system was already built with the feature of publishing customer contact information changes, this fraud-prevention feature can be built on top of the existing events. A new microservice is then built; it subscribes to `CustomerInformationChanged` events, checks the addresses against the central registry and if there is a mismatch, a `CustomerFraudSuspected` event is published and other (perhaps already existing) microservices process this event.

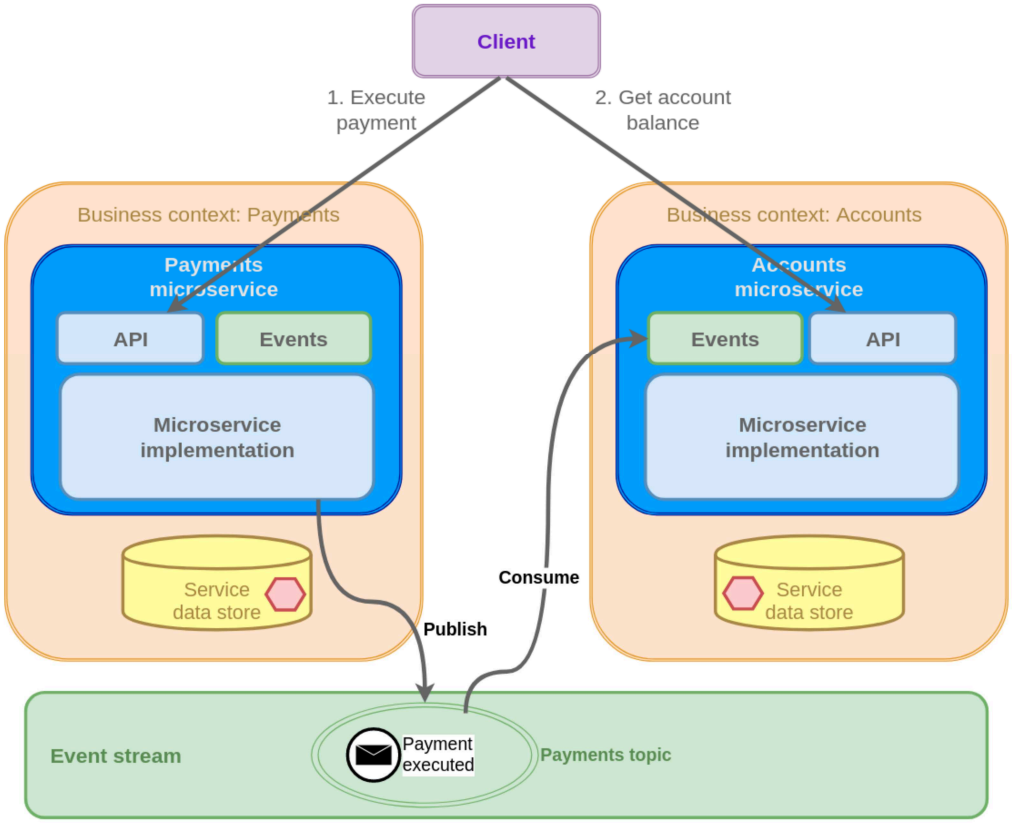

Figure 3.4: Propagating changes by publishing and consuming events.

3.3.1 Implementing events streams with messaging

Events are a particularly suited for the messaging pattern using a message oriented middleware (MOM) product. A MOM product handles storing pub-

lished messages and delivering them to subscribers. Some examples of MOM products typically used with microservices are RabbitMQ, Amazon Simple Queuing Service and Apache Kafka. But these products simply offer the primitive capabilities for sending and receiving messages. It is up the microservice system implementation to use them to form **event streams** where all changes in the system flow and from where any microservice can easily start receiving specific events – with the granularity that is suitable for both the data domain and non-functional requirements.

Some messaging products have features such as wildcard subscriptions that can be used to allow some freedom of granularity to the subscriber [11]. This enables the subscriber to choose at subscription time to be interested in e.g. login events from a specific country (subscription `"login.UK"`) or all login events (subscription `"login.*"`).

A key requirement for a messaging product used to form an event stream is that the product **must** support dynamic introduction of new consumers and producers [11].

3.3.2 Temporal decoupling

The system becomes more resilient to failures by communicating state changes and other new facts by blindly publishing events that any service can consume. As publishing an event and consuming it is not a synchronous but rather an asynchronous operation, all services handling the event do not have to be healthy when the event is published. If a service that is subscribed to the event is down, or uses, e.g., data stores or external integrations that are down, the event remains stored in the event stream. Once the consuming service is again in a healthy state, the event is fetched from the event stream and processing continues. [35]

To achieve temporal decoupling, it is **required** that the services can rely on the event stream to store the messages. As a consequence, only messaging solutions that support message persistence can be effectively used as the event delivery mechanism in an event-based microservice system.

As stated before, with synchronous integrations such as remote procedure calls, the availability of a business process is the product of avalabilities of microservices called in the process [27]. In practice no service can have a 100% uptime [27]. Assuming five synchronous microservice calls are needed to complete a business process and each service has a 99% availability (7 hours of downtime a month), the availability of the process is only 95% with 35 hours of downtime each month. Without temporal decoupling, a microservice system can lose both consistency (due to distributed non-transactional data stores) and availability. With temporal decoupling, only consistency is lost,

and the system can be designed to be **eventually consistent**.

3.3.3 Eventual consistency with events

There are two key properties that an event and event handler should strive to possess: an event should be **immutable** and handling the event should be **idempotent**. As an event represents something that has already taken place, an event should not be modified in any way once it is published. If the event should be corrected or even canceled, this should be dealt with by posting a new event that holds the new corrected state. This pattern is then an enabler for idempotent event consumers.

When consuming an event can be executed repeatedly and the end result is the same as executing it once, handling the event is considered idempotent [46]. If an event is immutable and handling the event is idempotent, the combination can be used to achieve **eventual consistency** between different parts of a distributed system. In an eventually consistent solution all state stores eventually converge to the same state, but before the convergence the states can differ [1].

As an example, let's say an e-commerce site built as a microservice system has a `Product` microservice and a `Search` microservice. When a new product is added to the system (via an partner integration, admin interface etc.) the `Product` service as the master data store for products processes the request. After successfully storing the product information the services publishes a `ProductCreated` event. As the `Search` microservice is subscribed to consume these events, the messaging product used for the event stream delivers the event to `Search` service. The service should now process the event by e.g. extracting full-text search tokens from the product description and storing these tokens in the services private data store (an Apache Solr search server) and finally acknowledge the event message as consumed so that is not delivered again. However, as there is no two-phase commit protocol in place, there is no way for the messaging product, microservice and Apache Solr datastore to achieve **exactly once** processing semantics for the message. But there are two other semantic options available. The service can first acknowledge the message and only then start processing it, giving the interaction **at most once** semantics. In this case, duplicate messages cannot occur but there is the possibility of permanent loss of consistency if the service crashes before the data is stored in Apache Solr. The other option is first processing the message and only when everything is completed successfully, the event message is acknowledged as consumed, opening to possibility of duplicate messages and giving the interaction **at least once** semantics. As the event is **immutable** and storing the extracted text to the search index is

an **idempotent operation**, duplicates of the event can always be processed safely. So, opting to use **at least once** semantics in this case guarantees that the `Product` and `Search` services data stores will **eventually** reach a consistent state.

3.3.4 Sourcing state from events

If services in a microservice system publish all changes to an event stream, the stream itself also forms a data store. The event history can also seen as state [3]. If all the persistent state a microservice needs is available from the event stream, the microservice itself does not necessarily need a separate persistent data store. When the service starts, it simply consumes the event history needed to build up the local state in memory. This pattern of data storage where the current state is not persisted, but built up from events that lead up to the current state is called **event sourcing** [30]. While event sourcing is a large and fairly complex subject that is not discussed in great detail in this section, here are presented some architectural design patterns (and pitfalls) that use elements of event sourcing and have synergy with event-driven microservice systems.

To effectively build services using the event sourcing pattern, the event stream is required to have some non-conventional features. As a service built with event sourcing is a consumer that at startup needs to access to **history** of events, the event stream must support "rewinding" the flow of events until some point in history. But the rewind support is also required with services that use regular persistent data stores when a new service is added to the system, but the service must initialize the data store from the event stream. If the event stream solution offers no support for the service to request this initial data load, each service must have an API for requesting an initial data set via some specific channel.

To elaborate the initial data set problem with the `Search` microservice example from the previous section, let us assume the microservice responsible for the full-text search and indexing features was developed months after the `Product` service was deployed to production. When the `Search` microservice is then deployed, it must form the search indexes for products that are already in the system. If the event stream supports this, the service can rewind the `ProductCreated` stream to the beginning, consume all events and thus complete the initial data load. There are however some caveats to consider in this design. First, this design requires that events do not expire and the full history is then always available. This potentially causes a huge amount of data to accumulate, taking of course storage space, but also forcing a backward-compatibility requirement on the event handler [30]. If the

`ProductEvent` was modified between service versions, the `Search` microservice will consume events with different schematics and all variations must be taken into account when developing the service.

In some cases, the events also can have identity, in the sense that an older event can be superseded by a newer one in the event stream [3]. This can be used to optimize the storage requirements needed for the event stream. If single a `CustomerInformationChanged` event carries all the information needed by consumers, it would be a valid design choice to keep only the latest event available in the stream history and drop the older ones. In this case the event identity would be, for instance, the primary key of the `Customer` entity, and the messaging product powering the event stream would be configured to drop messages with duplicate ids. In general, the retention requirements for different events can vary greatly in a system. Some events are needed only for a few seconds, others for days or weeks and some (such as audit trail events) might require being persisted forever.

If initial data load scenarios are rare and straightforward, it can be much simpler to develop simple batch jobs for loading the data between data stores than to use an event sourcing solution.

3.4 Synchronous commands and queries

Seeing all that can be done with events, should one use events also for sending requests, for example to execute a payment or to list all customers? An event is an asynchronous, immutable fact describing something in the past and it can be only used for publishing notifications. The execution of a payment is better modeled as a **command**.

A command is defined here a request to perform an action. Typically, the sender of the command also needs information on whether the command succeeded or not, and possibly with what results. Thus, unlike events, commands are often **synchronous** by nature.

Synchronous interactions create a tighter coupling between microservices than asynchronous ones and are less resilient [27]. But due to the tight coupling and explicit request-reply interaction, synchronous communication is also easy to reason about and tend to cause less complexity creep [27]. So both interaction strategies have their benefits and downsides. Thus, even with all the upsides of using asynchronous events listed in the previous chapter, there is little point in forcing all communication in a microservice system to the same model. If a response is needed for a command, and the command would be implemented with events, the system would have to build a synchronous action chain on top of asynchronous flow of events. To re-

duce global complexity, this should be avoided if at all possible. So, even though a command *can* be modeled with request-reply pairs of events (e.g. `PaymentRequested` and `PaymentCompleted`) [37], this is typically just abusing the event stream as an RPC mechanism.

As commands are usually synchronous by nature, they are especially suited for a synchronous integration technology. A microservice with a REST API for commands (such as "execute payment") and queries (such as "list payments") is a simple low-complexity solution. Publishing (and consuming) events from results of commands (see figure 3.5) makes the service a part of the whole, the microservice system.

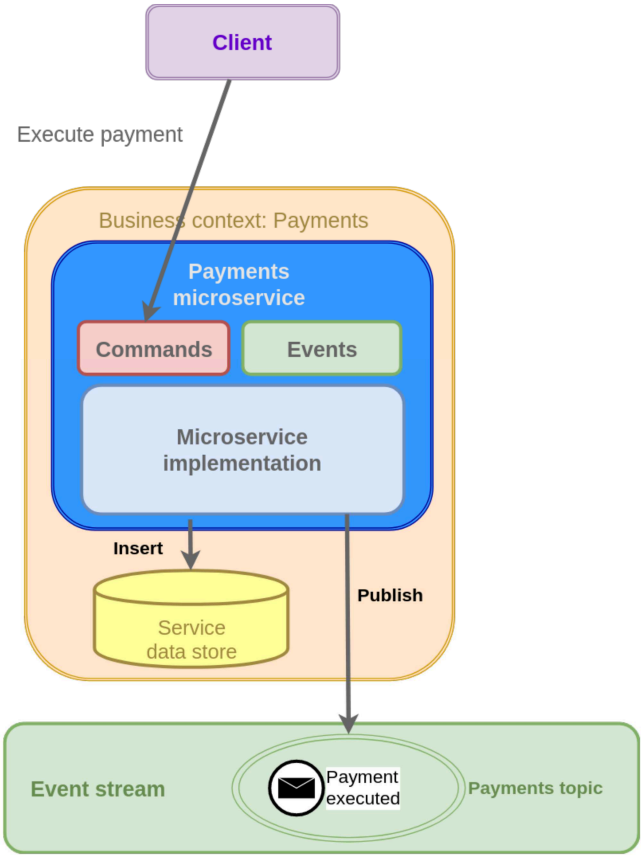

Figure 3.5: A microservice reacting to a command and publishing an event.

3.5 Apology oriented programming

The previous chapters presented a design for developing microservices that publish and react to events to form an eventually consistent data set. Such a design gives an architecturally and performance-wise scalable solution for handling state in a distributed system, and forms a structure for a system that is extendable and resilient to errors. However, such a design still struggles to meet reality of enterprise applications. Software systems (at least the useful ones) are not pure functions that only return an output based on some input. The reality is that all systems must deal with a great number of **side effects** which have not yet been taken into account at all.

A side effect is defined here as something that has an impact outside the system boundary [19]. Sending a customer an SMS is a side effect. Pushing 100 € bills out of an ATM is a side effect. Sending a bunch of trucks halfway across to world to deliver cargo is a side effect. If these actions are triggered from event handlers, processing the event is far from idempotent – by definition handling idempotent events do not cause side effects [18].

So, what can be done with side-effect causing events? The pattern presented the in previous chapters achieves eventual consistency by accepting the possibility of duplicate events, but duplicate side effects would typically be considered a defect by the business logic requirements. One option to try to avoid duplicate side-effects would be to persist a fine-grained status of the event processing by event identifier. In this case a microservice keeps track of the steps of processing the event in the data store. As an example, if a microservice consuming `UserCreated` events first saves the user attributes to the service-private data store, then calls an external integration to also add the user to a SaaS partner system and finally sends to user an email, the service would update the status of the tracked event row in the data store from `SAVED` to `ADDED_TO_PARTNER` to `EMAIL_SENT`. But this of course just changes the granularity of the duplicate side effects, as if the service crashes after sending the email but before updating the status, an duplicate email is then sent when processing the same event again. One can then try to make the granularity finer and finer, but as this is again a case of the Byzantine Generals problem there is no end to it. So, to put if briefly, there is no practical technical solution to this and no fancy consensus algorithm can fix the duplicate side effects. And here lies also the answer.

If there is no way to fix to software, one must change the business logic requirements, in this case give up on the **exactly-once** schematics. An event consumer can typically consume an event message multiple times in a **failure scenario**, i.e., an instance of the service crashed earlier while processing the

message. One often can add new special cased business rules on how the system is allowed to function in this case. How much the normal use case rules can be bent in a failure scenario then often depends on the business domain, e.g. a surgery robot has very different hard requirements than an online gambling application.

Apology-Oriented programming builds on this idea that it is often easier to ask for forgiveness than permission [4]. This may sound like a fairly drastic design pattern apply to potentially very business critical systems, but it actually matches surprisingly well with many real-world interactions and systems [4]. One example of Apology-Oriented programming is, surprisingly, a bank ATM. In a failure scenario, when the ATM is disconnected from the banking backend, a customer is allowed to do some withdrawals event though the account cannot be debited right away [4]. In this case, the ATM system has picked to favor availability over consistency in a failure scenario. This can also lead to an somewhat inconsistent state when the network connectivity is restored as a debit account can suddenly have a negative account balance – a state of affairs that was forbidden in the use case requirements. Another example of Apology-Oriented Programming is booking a flight. Due to loss of consistency, a plane can be overbooked, but the scenario is rare enough that airlines get away with it with apologies and by offering vouchers to unlucky customers [4].

Instead of going to extreme lengths to achieve consistency in a microservice system, both in side-effects and stored state, it can be a valid business choice to favor inconsistency in rarer scenarios. Choosing only two attributes from the CAP theorem triplet is not a system-wide or even service-wide decision and the choice can very well differ depending on the specific scenario [19]. If a duplicate side effect is mundane (such as sending a customer a duplicate SMS) or rare enough that it can be resolved by customer service agents, it can be a valid choice to create explicit business rules for these cases rather than try (and fail) to always build absolutely perfect software.

3.6 Composing operations in a microservice system

The previous chapters have presented a number of building blocks for developing isolated microservices. But a microservice system responsible for actually accomplishing potentially complex business tasks is **composed** from the isolated services that, following the Unix principles, *"do one thing and do it well"*. This chapter presents strategies for forming these composites. But

before composing is possible, the isolated microservices must define points of entry that can be used in the composite – **interfaces**.

3.6.1 Interfaces and contracts

An interface is a public entry point for a microservice. A service can have a number of different interfaces, one for each integration technology and interaction pattern. For example a microservice can expose a synchronous REST interface via a HTTP endpoint that takes commands and queries. The same service then exposes also an asynchronous event interface by publishing events to an event stream whenever a command is executed. Both these interfaces can be used by other microservices in the system. The provider of the interfaces and the consumers are then coupled. Thus a change in either of these interfaces can devastate on the system as downstream consumers of the interfaces can break due to an incompatible change.

To prevent breaking consumers, each interface should have a **contract** which defines the operations and the format of input and output messages for the interface. The contract can be anything from a formal specification to just a bunch of example messages, but the contract is what downstream consumers of the interface then use to design and implement the integration. As then the microservice providing the interface is evolved (by for example adding new attributes to messages), the contract must be evolved with it, optimally without breaking **backwards compatibility** [27]. But evolving the contract is hard or impossible if **forward compatibility** is not a part of the contract design [36].

There is a vast number of technical solutions for declaring explicit contract via languagues such as Protocol Buffers IDL[1], OpenAPI (Swagger), WADL, WSDL, XML Schema and so on. These formal specifications are used to generate client stubs or a server skeleton to reduce the amount of handwritten code needed for client-server communication [14]. Many of these technologies, especially when used naively, are a poor fit for backward- and forward compatibility requirements [14]. Often, if forward compatibility is not explicitly written into the contract as explicit extension points, generated clients can break even with changes that should logically be compatible [14]. This can be solved with versioned services and interfaces, but the versioning adds extra complexity in, for instance, service deployments and is usually not recommended [2]. What are recommended are patterns like Tolerant reader [14] and Consumer-driven contracts [36] where the consumer of the messages

[1]The solution to the contract language problem remains elusive as designs tend fluctuate. Now binary protocols are again hip and Protocol Buffers IDL looks quite identical to CORBA IDL from two decades earlier.

does **not** validate the message at all but just cherry-picks the attributes of the message that the consumer is interested in.

This principle of a lax consumer is often credited to John Postel who states in RFC 761 from 1980: *"implementations should follow a general principle of robustness: be conservative in what you do, be liberal in what you accept from others"* [33]. Robust consumers allow for flexible message formats. And flexibility in the message format then allows the microservice providing the interfaces to evolve, reduces tight coupling and makes composition less painful [37].

3.6.2 Choreography with events

In composing complex interactions from simple services, there are two main strategies for performing the composition: **choreography** and **orchestration** [29].

In choreography, composition is achieved by individual microservices reacting to events, performing actions and then publishing new events containing the outcome of these actions [29]. The business task is performed by this distributed network of microservices where each service is responsible for an isolated part of the process. There is no central command and control mechanism, no conductor driving the interaction of services, but the microservices are like dancers in a ballet, each reacting to other dancers around them [29].

As an example of a composed operation, let us assume that placing an order at an online store consists of the following steps:

1. A customer fills in order details with products and credit card payment information;
2. The credit card is debited;
3. Products are collected from warehouse to a postal package.

This interaction can be modeled with three microservices performing the steps with choreography: `Orders`, `CreditCardPayments` and `Shipping` service. When the `Orders` service receives the command for placing an order, the order is validated and stored in the services data store with the status attribute set to `NEW`. The service then publishes an `OrderCreated` event containing the order details and payment information. The `CreditCardPayments` services is subscribed to receive the `OrderCreated` events, consumes the event message and if a credit card number is present, calls an external integration to a credit card processor. If the external call succeeds, the processor has debited the credit card and the `CreditCardPayments` microservice publishes

a `PaymentCompleted` event. The `Orders` microservice consumes this event, updates the order status to `COMPLETED` and publishes a `OrderCompleted` event. The `Shipping` service then consumes the new event and creates a work task for the warehouse personnel to collect the products to a postal package.

This simple example highlights some key elements of composition with choreography.

First, microservices taking part in choreography are **loosely coupled** [29]. If a new channel for orders is added to the system (such as batch orders from large businesses), or another payment method is added, these changes can be implemented by adding new services to the system. These new services can then take part in the composed business process by consuming and publishing the same events as the existing services. As an example, if payments with PayPal are added to the system, a new `PayPalPayments` microservice is created. That service consumes `OrderCreated` events, processes events with a PayPal address in them, calls an external integration and finally publishes a `PaymentCompleted` event. Choreography as a composition strategy yields solutions that are flexible and amenable to change [29].

Second, despite composed business processes spilling over service boundaries, the microservices themselves are still **highly cohesive** [29]. Each service in the process is responsible only for those parts that are in its business domain. The `Orders` microservice is not aware of the steps required for getting the order payed, it only needs to know when the payment is completed.

Third, the business process implementation with choreography is **complex**. The interactions between the service are all asynchronous. As the choreography has no centralized service to drive the interaction, the only way to monitor, trace, or debug this asynchronous process is to try to keep track of the messages flowing in the event stream [29]. A complex process with a large number of steps fast leads to "event creep", especially when operations such as compensating actions or fan-outs are taken into account.

3.6.3 Service orchestration

Where in an operation composed with choreography the microservices act independently as dancers in a ballet, in **orchestration** the interaction is more akin to a conductor leading an orchestra [29]. In an orchestrated composition, there is a centralized point of control coordinating the different services [9]. The business process is implemented by an orchestrator calling different microservices in sequence and transforming and combining the results [9].

One way of implementing the online store example from the previous

chapter would be to make the `Order` service an orchestrator for the process. In this case, the `Order` service would, after receiving to command to place an order, first save to order to the service data store. The service would then synchronously call the `CreditCardPayment` service to execute the payment and update the order status in the data store to `COMPLETED`. After this the `Order` microservice would synchronously call the `Shipping` service to get the products packaged.

A number of observations can be made when comparing this implementation with the choreography implementation from the previous section.

The complexity of the implementation is much lower with orchestration. The interaction of the services is a sequence of simple synchronous calls and if the orchestrator knows immediately if the process succeeded or not [29]. The downside of this is the lack of temporal decoupling as the process can only succeed if all the microservices needed in the composition are available at the same time. The process as a whole is then less resilient to service and network failures.

With the implementation described above, the `Order` microservice is now tightly coupled to the `CreditCardPayments` service. If the interface of `CreditCardPayment` service changes, the `Orders` service must by changed as well and both services must then be deployed at the same time. Similiarly, adding new payment method microservices such as `PayPalPayments` also always causes changes in the `Orders` service due to the tight coupling.

The `Orders` microservices has also lost cohesion as it now is responsible for not only order but also payments and shipping. And if the business process becomes more complex, more interactions are added to the orchestrated composition and suddenly the `Orders` microservice connected to a dozen of other services. Business logic of all sorts accumulates to the service that was supposed to be responsible for only orders. This can lead to the antipattern of "god" services where the microservice system consists of a few bloated services directing a large number of anemic logicless CRUD-based services [29].

So this design for the example process orchestration is obviously far from optimal. However, cohesion can be restored and coupling moved out of the `Order` microservice if the role of the orchestrator is given elsewhere. Orchestration as a solution becomes more manageable by making orchestrators separate services. One option is moving the orchestration logic closer to the client to a frontend aggregate [4] (see Section 3.6.5), but also business process-specific dedicated orchestration microservices are a valid option. Keeping the orchestration logic out of regular microservices helps services maintain cohesion and contains change related to the orchestration to the business process-specific orchestrator service.

Both orchestration and choreography have their benefits and downsides as composition strategies. These strategies can also be combined to form hybrid solutions that have both synchronous commands and event-based asynchronous interactions [29]. Both strategies have their uses and the picking either should depend on the requirements of the specific use case to be implemented.

But regardless of the strategy choice, the implementation has also to take into account that a step in a composed operation can (and will) fail and often the changes already performed should then be **compensated**.

3.6.4 Compensation with Sagas

In a microservice system, the number of moving parts that can fail is typically much larger than in a monolithic application. All microservices and integration between them can and eventually will fail in a myriad of ways. Taking the possibility of failure into account is a crucial part of designing a resilient microservice system [27].

Considering the order placement business process from Section 3.6.2, the process can fail in multiple steps. For example, the credit card used for payment may not have sufficient credit available. Or the payment succeeds but in the shipping step the ordered products are no longer available. In addition to these business failures, the system may experience technical failures, for example the external integration between the payment service and the credit card process may be broken. Any of these failures might abort the business process and might leave the system in an inconsistent state if already performed changes are not undone.

Distributed transactions handle the uncertainty of transaction completion trough the use of long-lived locks on data [27]. In contrast, the **saga pattern** handles this uncertainty with compensating workflows [27]. Instead of conceptualizing a business process as a single long-lived distributed transaction, the process is seen as a sequence of multiple local transactions [5]. In a saga, each local transaction is paired with a compensating transaction [16]. When the compensating transaction is run, it reverts the changes performed by the original transaction (see figure 3.6) [16].

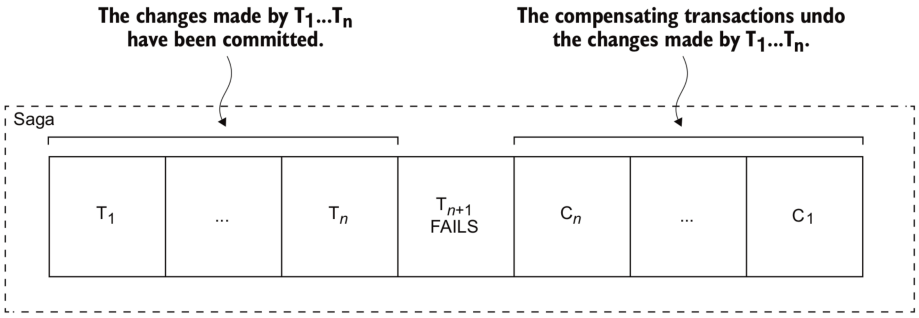

Figure 3.6: If a step in a saga fails, all the preceding completed steps are undone by executing compensating transactions [35].

Sagas can be used with both choreography and orchestration [27]. In an choreographed saga, a failure in a step of the saga is signalled by a participating microservice by publishing an event [27]. Referring to the order placement orchestration example from Section 3.6.2, the `Shipping` service might send a `ProductsNotAvailable` event if the ordered products can not be packaged. The `CreditCardsPayments` microservice would then consume this event and perform an appropriate compensating action per the use case business logic. In this case the compensating action might be reimbursing the credit card payment.

In an orchestrated saga the orchestrator microservice is the coordinator for the saga [27]. The coordinator calls each performing microservice in sequence to perform a step. If a step fails, the coordinator then calls the compensating operations for each step already completed (see figure 3.7) [27]. In an orchestrated saga the coordinator is a finite state machine where the result of each microservice call triggers a state change [27]. Eventually the state machine ends with either in a successfully completed saga or an undone saga.

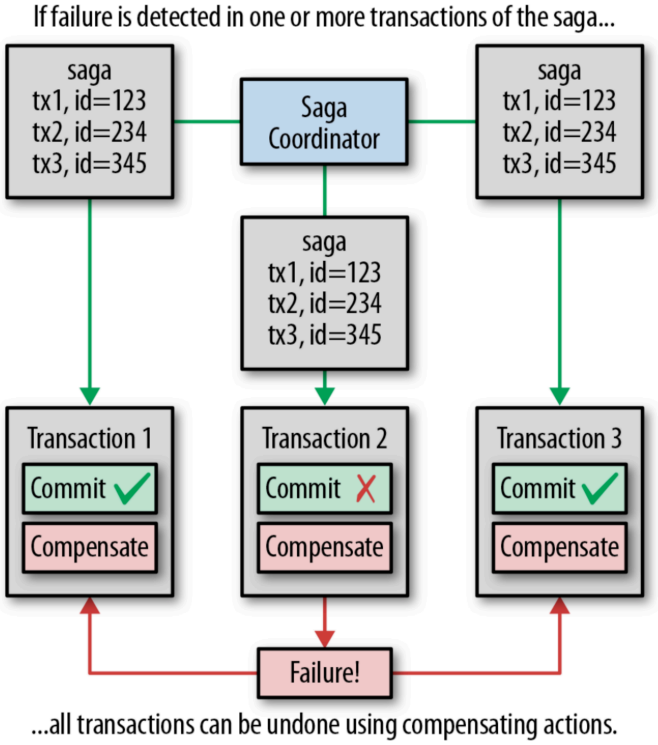

Figure 3.7: A saga coordinator is responsible for calling compensating actions if a transaction fails [5].

It is worth noting that a compensating transaction does not necessarily revert changes in the service private data store. The compensation is a semantic operation and it can by design leave the data store in a different state than it was before the execution of the saga [16]. In the case of the order placement example, if a payment fails, the order might not be removed from the `Orders` microservice data store, but the status for the order might be updated to `FAILED`. Similarly, a saga can "rollback" side effects with compensating actions. For example, if a step in the saga has already sent a customer an email message, the compensating transaction might send a new email with apologies [27].

3.6.5 Frontend-specific aggregates

When building composed operations with orchestration (see Section 3.6.3), the orchestration logic can (and should) be placed in a separate microservice. The client needing to perform the composed operation can then simply

call the orchestrator microservice without being aware of the composition specifics. But in addition to orchestrating commands, an orchestrator service can combine and transform responses from queries. Hence, an orchestrator service can be built that offers composed commands, composed queries and transformed responses that better suit a client. This service is an aggregate for a client, a **backend for frontend** (BFF).

A backend for a frontend offers a single unified interface for one specific client [7]. A microservice system can have a number of backend for frontends, one for each client type [7]. Some examples of clients are a customer-facing web application, a mobile application for customers or a administrative interface for company personnel. Each of these clients use different composed operations, start different business processes and require data at a different granularity. Each backend for frontend has an interface customized according to each clients needs [7]. The BFF can for example filter out attributes that are not used by the client and translate and combine results of queries into channel-specific representations that better fit the requirements of the client [7].

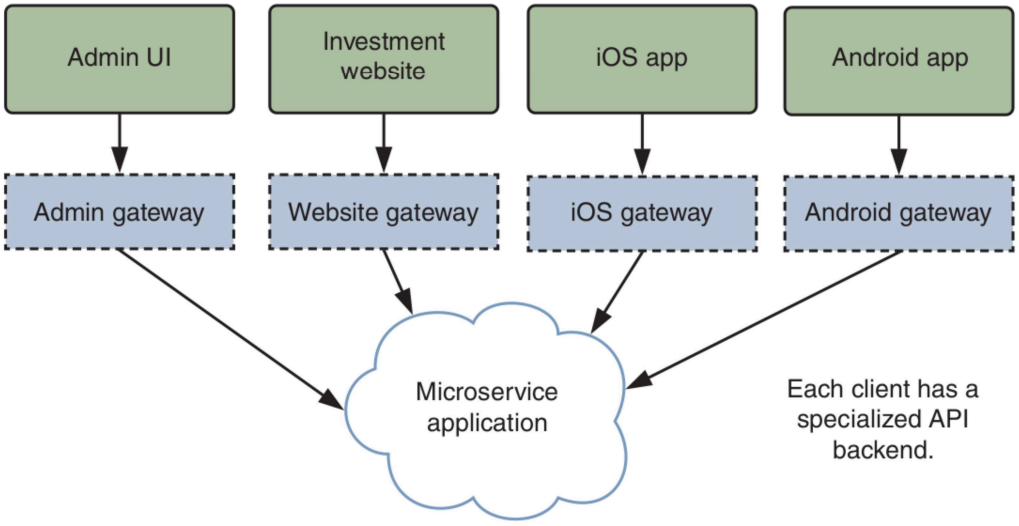

Figure 3.8: Clients and client-specific backend for frontend gateways [35].

The backend for frontend also allows for a point for control for each client type. The BFF can present a stable interface to the client, so that changes can be made in microservice interfaces without the change requiring, for instance, releasing new client application versions.

A related pattern to backend to frontend is an **API gateway**[2] [35]. This can be seen as a single backend for frontend for all clients. An API gateway can contain similar functionality as a backend for frontend, but it is typically more focused on cross-cutting edge concerns such as authorization, rate limiting, caching and metrics [35].

The two patterns presented here can also be combined. The system can have an API gateway responsible for edge concerns which then routes to a number of backend for frontends responsible for client specific composition and transformation [29].

3.6.6 Data integrity in composed operations

When composing operations in a microservice system, services often process entities owned by other services. For example, an `BankAccounts` microservice might consume a `PaymentExecuted` event that carries a `Payment` entity. Or a backend for frontend might orchestrate an operation where first an `Account` entity for a customer is queried from an `BankAccounts` service and then a `Payments` service is commanded to execute a payment from that account. But this poses a data integrity problem. If the payment backend can transfer funds from **any** account and command is executed with the credentials of a **specific** customer, how can the `Payments` microservice be sure that the customer is authorized to pay from the account? The `Payments` microservice is not the owner of the `Account` entity so it does not have access to the master data store for accounts where the authorization rules are stored. The **Payments** service can of course make a synchronous query (for example a HTTP GET) directly to the **BankAccounts** microservice to check whether the customer is authorized to pay from the account. But this is a inter-service synchronous integration that tightly couples the services. And it also adds another (hidden) synchronous call to the orchestration which as a whole can perform poorly if the amount of synchronous calls gets too large.

One solution to the payment integrity problem is to create a persistent cache of account authorizations in the `Payments` microservice private data store. In this case, the `Payments` service would subscribe to events published by the `Accounts` microservice (such as `AccountCreated`) and store the account authorization in the service data store. Then when a command for executing a payment is received, the `Payments` service can check the autho-

[2]The term API gateway is today often a marketing term used to sell a products with very varying capabilities with the promise of delivering the microservice promise. This is similar to the SOA days when companies often "went SOA" by buying an Enterprise Service Bus (ESB) product. Actually, some API gateway products contain the exact same functionality as previous generation ESBs.

rization from the data store. This is a performant solution, but it does carry the overhead of requiring additional data storage. If the same integrity check is required in dozens of microservices, the overhead can be non-negligible.

A more lightweight solution for guaranteeing integrity for an entity in a composed interaction is to combine the security-sensitive attributes of the entity together to a **token** and attach a cryptographic signature to it. This token is then added as an attribute to the entity. In the case of the payment example above, one can combine the account number and authorized customer ids as claims in a **JSON Web Token** (JWT). A JSON Web Token is a bag of attributes (called claims) with an attached signature [6]. When using asymmetric cryptography, one can then set up a scheme where only the owner of an entity, in the example case `Account`, holds the private key needed to sign the specific token. Other microservices have access to the corresponding public key needed then to verify the authenticity and integrity of the token [6]. In the example above, the `Payments` microservice then does not take a source account number in the payment command, but a signed token that the service can verify. The downside of the signed token solution is then the need for key management and the solution works best if the microservice system already has a centralized key management application in place.

3.7 Summary

So far the thesis has presented a model for a microservice system consisting of isolated, uncoupled microservices that store data in private data stores, publish events whenever their "world" has changed, consume asynchronous events from other microservices and accept synchronous commands and queries. The microservice system is in a constant flux of state change but all changes eventually lead to a consistent end state. The system can execute complex business processes asynchronously with choreography or synchronously with orchestration and the processes can support rollbacks with the use of the saga pattern. Data integrity in the processes can be guaranteed with signed tokens. The microservice system can offer client-specific customized stable interfaces with the use of orchestrating backends for frontends.

To gain something, one must let go of something else. On the path from a single monolithic application to a distributed microservice system we have given up consistency and centralized control but gained an architecture that is the sum of independently evolvable parts. The next chapters present how such a system can be built in the public cloud.

Chapter 4

Microservice systems in the public cloud

This chapter presents building blocks for developing microservice systems in the public cloud by using Amazon Web Services as a representative scenario. The building blocks are chosen to fit the design patterns and architectural choices presented in the previous chapters and the goal is to form a blueprint for event-driven microservice systems in the public cloud.

From among possible options, Amazon Web Services was selected for the study as it is the prevalent option for a cloud platform. Other possible choices would have been Microsoft Azure and Google Cloud Platform.

4.1 A Cambrian explosion

When designing a system architecture on a proprietary platform such as Amazon Web Services, one is tied to the features offered by the platform and constrained by its shortcomings. However, the problem with AWS is not that the feature set of platform would be lacking, but that the feature set is so comprehensive and contains so much overlapping services that one is easily lost. With thirteen years of history [22], in 2019 the Amazon Web Services service palette contains over 150 different services. Dozens more are released every year. When designing a microservice system on AWS, one is confronted with a nearly endless variety of choices for managed data stores, messaging systems, runtimes, configuration management and so on. All of these choices have their own niches where that specific service excels and all the choices have their range of applicable usage scenarios.

Making architectural choices for a system is always about making tradeoffs. Some requirements are given more importance than others, but all re-

quirements must still be fulfilled. In this thesis, emphasis is given to creating a reference architecture for a microservice system on Amazon Web Services that is general enough to suit **the common case**. Services are chosen from the AWS service palette with the goal of being **good enough** for the common 80% of design decisions. The goal is not to design a microservice system that has the best performance in all cases or is the most scalable or resilient to failures. The goal is to find simplicity amidst the complexity of today's cloud architecture and to present a blueprint for a "boring" cloud computing solution that gets the job done.

Figure 4.1: A subset of Amazon Web Services platform services from `https://console.aws.amazon.com`.[2]

[2]If one picks Elastic Container Service (ECS) as the microservice runtime and Relational Database Service (RDS) as the data store, one then is confronted with choosing a

4.2 Microservice runtime platform

The Amazon Web Services platform services suitable for a microservice runtime can be categorized as **infrastructure as a service** (IaaS) or **platform as a service** (PaaS) [25].

Infrastructure as a service allows for on-demand scaling of the infrastructure [44], but all the management burden of such as configuration and installing security updates, for instance, lies on the development team. Amazon EC2 is an IaaS service which offers unmanaged virtual machines (called instances) [21], where one can deploy microservices as classic Unix deamons. But as microservice architecture is generally suitable only for large-scale systems [28], using EC2 as runtime[3] would require building automation solutions for deploying, updating and monitoring all the daemon processes scattered over dozens of instances.

Unlike EC2 and other IaaS AWS platform services, platform as a service solutions have the required automation capabilities built in [21]. Using these solutions reduces the operational overhead of running, maintaining and monitoring individual microservices [21]. Amazon Seb Services offer a range of PaaS platform services for deploying applications and functions such as Elastic Beanstalk, Elastic Kubernetes Service, Elastic Container Service. There is also now a Function as a Service (FaaS) option available in the form of AWS Lambda. So which to pick as candidates for a generic microservice runtime?

If Amazon EC2 offers bare virtual machines without any restrictions, the other end of the spectrum is Elastic Beanstalk where only limited application types can be deployed. In this respect, Elastic Beanstalk is similar to first-generation PaaS solutions such as Google App Engine[4] or Heroku[5]. An overly constrained runtime is a poor choice for microservice architecture where each microservice can have very varying characteristics and requirements. So the runtime platform should support both short-lived and long-lived tasks, web applications as well as services with only event-based

ECS launch type and RDS engine type pair from the total of 14 different combinations. Whatever the choice, it can be paired with at least six top-level AWS services that can act as message-oriented middleware with varying features for delivery guarantees, message persistence and subscription types. To route HTTP traffic to the microservice, one then has to choose from three different Elastic Load Balancer types. So a service that runs on ECS, stores data in RDS, has an HTTP interface and publishes events can be implemented with at least 2^8 different AWS service combinations.

[3] When running more than one microservice per EC2 instance

[4] `https://cloud.google.com/appengine/`

[5] `https://www.heroku.com/platform`

interfaces. Moreover, the technological heterogeneity commonly used in the microservice architecture should be supported [28]. To enable the deployment of varied microservice implementations, the Amazon Web Services runtime service should support **containers**.

4.2.1 Containers and Docker

Amazon EC2 allows for great flexibility in application design by making the entire virtual machine the unit of deployment. However, in a microservice system this creates significant overhead as a single microservice is usually a small deployment package (in the megabytes to tens of megabytes range) with fairly modest requirements on CPU and allocated memory. **Containers** allow for the flexibility of a virtual machine deployment with the overhead of an operating system process.

A container encapsulates an application and its dependencies as an executable artifact [27]. Starting the container merely starts a regular operating system process with an isolated file system, process space and capabilities (see figure 4.2) [10]. As containers are processes running on the operating system kernel of the host they have a limited overhead, thus, several of them can run on the same machine [27]. The containers are isolated from each other and can have defined quotas for resources such as memory [10].

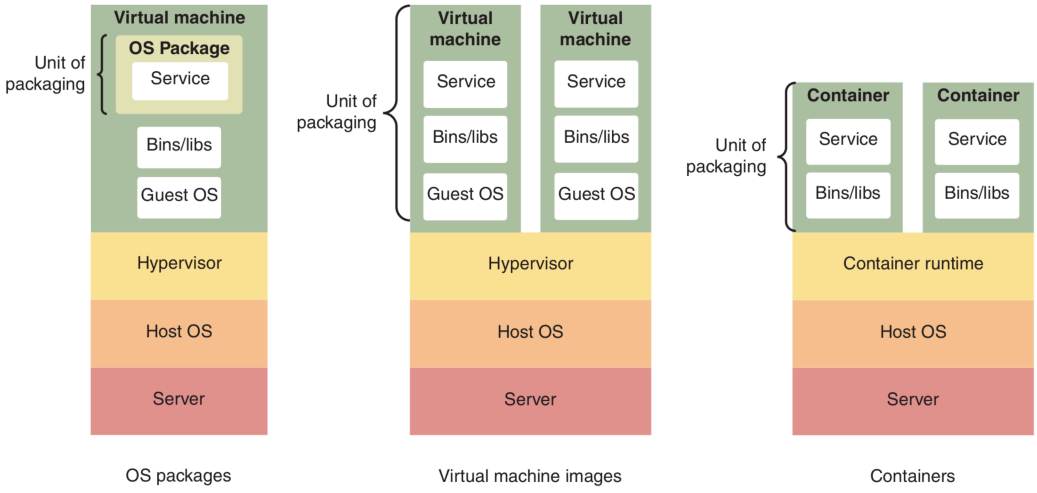

Figure 4.2: Application packages, virtual machines and containers as deployment artifacts [27].

Docker is a user space daemon for running and building container images

[20]. Docker enables building individual microservices as Docker images that are then uploaded to a Docker image registry [20]. The images are portable in the sense that they can be executed on any host where the Docker daemon is running. In an optimal situation, this allows a software developer to run a Docker image on a local development machine and then deploy the same image to a cloud service where it runs with identical results [20]. In many cases, container technologies such as Docker have been one key enabler for the adoption of microservice architecture [20].

When the deployable artifact for a microservice is a Docker image, the microservice system is not constrained in the choice of programming language, used libraries or tools. Making the image as the lingua franca of the Amazon Web Services platform runtime allows one to create portable consistent deployments using the technology stack that is best suited for the problem at hand.

4.2.2 Elastic Container Service

The AWS service palette offers two services to run Docker images for microservices: Elastic Kubernetes Service (EKS) and Elastic Container Service (ECS) [21]. Both of these a managed Docker environments and tasks such as installing, operating and scaling the cluster management infrastructure are handled by AWS [21]. Both services are a valid choice for a microservice system runtime, but if the development team is not already familiar with Kubernetes (an open-source platform for container management, a complex software in its own right[6]), Elastic Container Service is a simpler choice.

Elastic Container Service is an AWS platform service that manages running Docker containers on a cluster of hosts. ECS is responsible for distributing container deployments in the cluster and matching deployment task definitions specifying the required computing and memory resources with the available capacity on each host. Containers are deployed to ECS as images from a private Docker image registry, the Elastic Container Registry (ECR) (see figure 4.3) [47].

[6] `https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/`

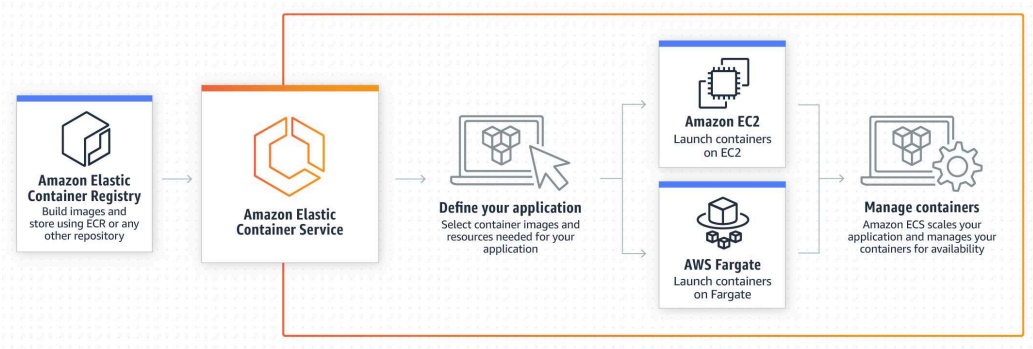

Figure 4.3: Application lifecycle from a Docker image in Elastic Container Registry to a running container.[7]

For a microservice system, Elastic Container Service provides the necessary target runtime for the individual microservice containers. Multiple containers of the same individual microservice image can run on multiple different hosts – both the number of hosts and the number of containers can then be scaled depending on the load [47]. This prevents a host or a container becoming a single point of failure in the system [37].

Task definitions in ECS allow for a deployment to specify required resources such as the amount of memory or computing capacity [47]. But microservices in a system can have different characteristics on more levels than just on the raw capacity needed by the service. The microservice architecture encourages creating separate services for non-cohesive business or technical capabilities [27]. This means that some services in a microservice system receive traffic 24/7 while other might only run a batch job loading external data or creating reports nightly, weekly or monthly. An external trigger such as received batch data may suddenly require a microservice to scale from a few running containers to hundreds. To match these different requirements, there are two different ECS launch types that can be used.

EC2 launch type of the Elastic Container Service deploys containers on Amazon EC2 instances that are part of the cluster [47]. Containers are then deployed on these virtual machine instances. The number of EC2 instances can be scaled according to load, but it is up to the development team to automate the scaling and manage the instances. This launch type is best suited for microservices where the capacity requirements are fairly stable.

For deploying microservices where the needed capacity can vary greatly (such as computationally intensive scheduled jobs), the **Fargate launch**

[7]Image source: `https://aws.amazon.com/ecs`

type is more suited. Fargate is a **serverless** deployment model where the cluster infrastructure is abstracted away and is completely managed by AWS [47]. Scaling a microservice on Fargate is not constrained by the bounds of EC2 cluster resources but can be scaled automatically based on load to hundreds of container instances.

Elastic Container Service is the basic building block for a microservice in the architectural blueprint described in this thesis. ECS abstracts the detailed mechanics of running microservice containers but allows for microservice deployments with highly varying characteristics.

4.3 Implementing event streams

In an event-driven microservice system, the event stream is the backbone of of the system. As such, how to implement an event stream is one of most the most crucial design decisions when building the system architecture.

As described in Section 3.3.1, event streams are particularly suited for a message oriented middleware product. There is no shortage of messaging solutions, in Amazon Web Services, so a choice of the AWS platform service must first be made. For it to be considered a candidate for an event stream implementation, the AWS messaging solution must be able to support the use cases presented in Section 3.3. Specifically, the messaging system must the able to fulfill the following requirements:

1. Publishers, consumers and message destinations must be **dynamic**. A message publisher does not need to know the consumer(s) of the message and a new consumers can be added while messages are already being published. As described in Section 3.3, one of the main benefits of the event-driven pattern is supporting the dynamic nature of microservice systems. In a microservice system, new features can be implemented by introducing new services instead of modifying existing ones [37].

2. A published message must be able to be consumed by **multiple consumers**. This makes the semantics of the message destination a publish-subscribe topic instead of a point to point queue [42]. As mentioned in Section 3.3, microservices in an event-driven system blindly broadcast all state changes as events to the event stream. A single event is then typically consumed by more than one other microservices.

3. A published message must be **persistent**, i.e. the messaging system must store messages instead of just forwarding them. Temporal de-

coupling presented in Section 3.3.2 requires that events are stored in the event stream even when the consumer microservice is not available. Also persistent messages give a slightly better guarantee of eventual consistency described in Section 3.3.3, in the case the messaging system itself crashes.

4. A message must be considered delivered only after is has been **acknowledged**. Sending a message to a consumer does not indicate a successful delivery and the consumer must be able to explicitly acknowledge the message. The *At least once* consumer semantics used for achieving eventual consistency described in section 3.3.3 require that the consumer can decide when event handling is considered completed.

5. For event sourcing and initial data load scenarios, a consumer must be able to **consume messages it has already acknowledged**. Sourcing state from events as presented in Section 3.3.4, requires that a consumer can build its (initial) state by consuming a series of events from the event stream. The state is not necessarily persisted, in which case the same events are consumed again the next time the services loads the state [30].

With an event stream implementation that fulfills these requirements, all the event-related patterns from Chapter 3, such as composing operations with choreography and choreographed sagas, can be used.

4.3.1 Alternatives for a messaging backbone

The simplest option for a messaging solution in Amazone Web Services is the aptly named Simple Queue Service (SQS). However, SQS offers only single producer - single consumer queues and not publish-subscribe topics which are required for event streams. To achieve topics semantics with SQS, the service can be combined with another AWS platform service, Simple Notification Service (SNS) [21]. SNS offers topics to which one can subscribe SQS queues, effectively achieving one to many publish-subscribe model. But the combination of SQS and SNS does not fit the other event stream requirements very well. In fact, such a combination is not very dynamic as it requires registering each SQS queue explicitly to the SNS topic, thereby creating overhead in deployments and in managing the queues. SQS also does not make message redelivery possible after it has been acknowledged, so to support event streaming the SNS topic would have to be also combined with a persistent data store.

Jung et al. [21] present a solution for building an event stream with Amazon Kinesis (see figure 4.4. Kinesis is a managed AWS platform service that does real-time processing of large data streams [48]. While Kinesis itself does not fulfill the requirement set of a microservice system event stream, the combination designed by Jung et al. does offer most of the required features. The Kinesis event stream design consists of a multiple AWS services, namely Kinesis Data Streams, Kinesis Firehose and Simple Storage Service (S3) [21]. Publishing an event from a microservice is performed by writing a message to a Kinesis Data Stream [21]. The Data Stream is read by a Kinesis Firehose that persists the event message to S3 storage [21]. Microservices can consume events by constantly polling the Kinesis Data Stream [21].

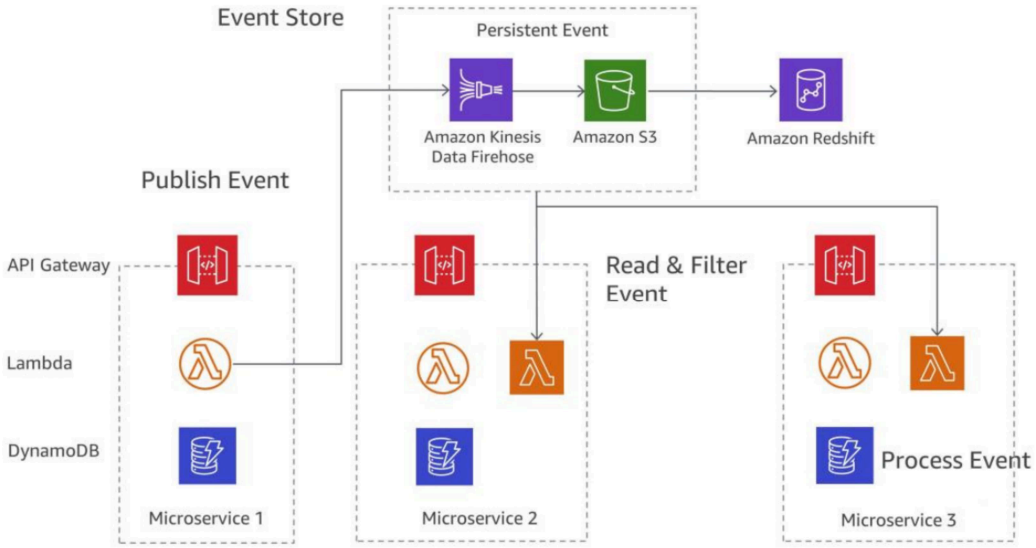

Figure 4.4: An event sourcing implementation on AWS using Kinesis Data Streams [21].

The Kinesis event stream pattern matches the requirements set for the event-driven microservice system event stream. Kinesis Data Streams offer publish-subscribe semantics with support for multiple consumers [48]. Events are persisted to S3 from where a custom lambda can be triggered, for instance, to load events in an event sourcing scenario. Consumers for the Kinesis Data Stream support the concept of a checkpoint for keeping track of processed records [48], and this can be used achieve *at least once* processing semantics. The Kinesis event stream pattern is a solid solution. But, as it is composed of multiple independent AWS platform services, it does require

some custom glue connecting the different services and for handling features such as initial data loads.

But in the AWS platform service palette there actually is a single managed service that also fits the requirements from section 4.3, **Amazon Managed Streaming for Kafka (MSK)**.

4.3.2 Propagating events with Amazon MSK

Amazon Managed Streaming For Kafka (MSK) is a fully-managed AWS platform service for Apache Kafka message brokers [49]. Kafka is a low-latency publish-subscribe messaging system designed for big data stream and distributed environments [45].

Kafka offers streams with multiple consumers, persistent messages, explicit acknowledgements; most importantly, Kafka streams used in MSK are rewindable [45]. As the consumer of a Kafka stream can start consuming the topic from any point (e.g., from the first record, last record or at an arbitary offset), the topic itself becomes a persistent data store [45]. This makes is possible to treat stream consumers as just a generalization of the batch processing model [45], exactly what is needed for implementing event sourcing and initial data loads.

With Apache Kafka, producers send messages to topics from which 0-n consumers can consume the same message [52]. Unlike most other message oriented middleware solutions, Kafka makes no distinction between a queue and a topic, but the semantics of the message delivery can be chosen by the consumer. Each Kafka consumer is part of one consumer group and each message in a topic is delivered at least once to each consumer group [52].

The flexibility offered by Apache Kafka consumer groups is extremely useful for a microservice system event stream. Building a microservice system is an iterative process, with the system gaining new features by adding more services to the network of individual microservices [37]. When the microservice publishing an event is built, the is no knowledge on which microservices in the future will be consuming those events. Similarly, the precise semantics of consuming the event message are entirely up to the needs of the consuming microservice. One service can consume the event to update state in the services persistent data store. In this case, the message needs to be delivered to only **one running container** for that microservice. Another microservice might consume the event to evict in-memory caches, in which case the message needs to be delivered to **each running container** for the service. With Kafka consumer groups, the behaviour model is chosen by the consumer (and not the broker or topic configuration). When all container instances for a microservice share the same consumer group id, each message

is delivered to only one container. And when each container has a unique group id, all containers for the microservice receive each message.

One of the benefits of microservice architecture is the ease of horizontal scaling, i.e. adding more capacity to the system by cloning more copies of a microservice [35]. For horizontal scaling to be effective with event-driven microservices, the messaging system needs to be able to distribute messages among multiple consumers. With Apache Kafka used in Amazon MSK, this is achieved by splitting a topic into multiple partitions [52]. Within a consumer group, each partition of messages is sent to only a single consumer, but a consumer can concurrently receive messages from multiple partitions [52]. When the amount of consumers in a consumer group changes, for example, when scaling up the amount of container instances in Fargate, the Kafka broker will redistribute the partitions for all the consumers [52].

Using Amazon Managed Streaming for Kafka as the event stream implementation in an event driven microservice is a scalable solution requiring no custom integrations or IaaS virtual machines to manage. Amazon MSK fulfills the requirements set for an event stream implementation technology set in Section 4.3 and, as a single managed AWS platform service, it is in line with the design goal of finding a simple solution as set in Section 4.1.

4.4 Implementing frontend aggregates

As described in Section 3.6.5, a client typically interacts only with a frontend aggregate instead of exposing individual microservice interfaces directly to clients [44]. The aggregate presents a stable API to the client and can compose and transform responses from microservice to better suit the needs of the specific client.

For building frontend aggregates with HTTP APIs, Amazon Web Services offers the API Gateway platform service [50]. In an API Gateway, HTTP resources for different microservices are defined and the gateway routes traffic to the services based on the incoming request [50]. The gateway can handle authentication and authorization of API calls, rate limiting and also run data transformations on the replies for backend microservices [50].

This allows most of the functionality required for the backend for frontend pattern presented in Section 3.6.5. So the obvious approach seems to be to create an API Gateway frontend for each client. If the gateway is a just a thin reverse proxy, this is a viable choice. But one of the responsibilities of the backend for frontend is to present an interface customized for the client. This often requires creating API endpoints which return a coarse grained responses which are composed from the responses of multiple microservice

backend calls. But an AWS API Gateway can only do simple transformations [50] and must delegate composing logic to an upstream component. AWS API Gateway is best suited for features that can be defined as **edge concerns**. This includes responsibilities such as authorization, caching and route limiting that need to be handled at the outer edge of a system.

While an AWS API Gateway is a suitable choice for handling edge concerns [50], for building a backend for frontend there is no managed AWS platform service available. A backend for frontend can be responsible for complex logic such as composition, orchestration and stateful orchestration as with sagas as presented in section 3.6.4. So a BFF must be built from scratch using a number of different AWS services.

One solution for building a backend for frontend is combining the API Gateway, Elastic Container Service and Application Load Balancer platform services with custom code as detailed next.

4.4.1 Backends for frontends with API Gateway, Elastic Container Service and Application Load Balancer

When building a backend for frontend with Elastic Container Service, a microservice running in an ECS container acts as a gateway between the clients and the backend services. The gateway microservice can be built as a lightweight web application that accepts requests from a single client type such as a web application running in a web browser, calls one or several backend microservices and returns results transformed to suit that client type [44].

For the BFF microservice to be able to call the backend services, the target services must not only expose interfaces, but the caller must be able to discover the interface endpoints. An Elastic Container Service cluster can have dozens of virtual machines each running some subset of the containers deployed on the cluster. But the caller needs to locate the endpoints available for the specific service that it wishes to call. One pattern that can be used to solve this discovery problem is the **service registry** [27]. In this pattern, each microservice registers the network endpoints it is available at in a centralized registry [27]. A client that wishes to call the service then queries the registry for available endpoints [27]. This is a very flexible approach supporting complex use cases such as releasing newly deployed services to a controlled subset of clients [27]. But the service registry approach also adds complexity the system in the form of an additional runtime component that must be monitored and maintained.

In this reference architecture, flexibility is traded for simplicity and the emphasis is on using AWS managed services where feasible. A simpler solution for the discovery solution is to use a **load balancer** [31]. In this pattern, instead of a calling microservice querying a service registry for the endpoints of the callee service, the caller can blindly send the request to a load balancer. The load balancer then has the knowledge of the endpoints for the target service.

For load balancing HTTP requests to microservices, instead of using a classical load balancing solution such as HAProxy, Nginx or Apache HTTP Server, one can use an AWS managed service called **Application Load Balancer** (ALB). When a client sends a request to an ALB, the load balancer routes the request according to the defined routing rules. When the request matches a rule, the target microservice is selected from a routing target group. For each microservice in a target group, the load balancer checks the status of the service by periodically polling a health check endpoint. If the target microservice does not report a healthy status, it is removed from the ALB target group. But this is where combining AWS managed services pays off, as in addition to removing the faulty microservice from the load balancer target group, the container is also shut down in Elastic Container Service and a new container started. When new microservice containers are deployed to to ECS, the containers are then automatically registered to an Application Load Balancer target group. [21]

An example microservice system architecture with a backend for frontend built with API Gateway, Application Load Balancer and Elastic Container Service consists of six layers.

1. API Gateway: responsible for edge concerns such as API key authorization, routes HTTP requests to the BFF ALB.

2. Application Load Balancer for BFF, routes HTTP requests to BFF microservices.

3. Elastic Container Service cluster for BFF: responsible for running BFF containers.

4. BFF microservice: Performs tasks by calling a number of other microservices over HTTP.

5. Application Load Balancer for microservices: routes HTTP requests to the services in the microservice system.

6. Elastic Container Service cluster for microservices: responsible for running service containers.

In the example architecture, the backend for frontend microservice can orchestrate calls to different microservices and combine the results of those calls to suit the needs of the client. As the BFF is merely another microservice, a lightweight web application, the architecture offers flexibility to choose to implement functionality in either in backend microservices or in the BFF service. But the solution does have its limits. Backends for frontends and other gateways typically do not store persistent state [44]. And as such, if the BFF has no capability to orchestrate stateful processes, it is mostly limited for performing composition of API calls. For orchestrating complex business processes, solutions are needed that support persisting the process state.

4.5 Implementing orchestrated processes

As presented in Section 3.6.2, if a business process is implemented with orchestration, a central coordinator is required for calling different microservices and acting on the results of those calls. In a microservice system that relies heavily on publishing and consuming events, some of those calls often are asynchronous so that the coordinator will move to business process forward based on the events it consumes from the event stream. This requires keeping track of the process state, so that the process can continue once the asynchronous operation has completed. Similarly, a process might have steps that require human interaction, e.g., a process for granting loans might be fully automated unless the loan sum exceeds a certain amount and must then be manually approved. Once the approval is given, the automated process again continues execution.

Coordinated processes, especially long-running ones, can benefit from building the processes on top of runtimes that offer support for elements often present in business processes such as temporal decoupling, retries and compensating actions. This chapter presents two alternative solutions for building coordinated processes in Amazon Web Services.

4.5.1 Processes with AWS Step Functions

AWS Step Functions is a managed AWS service for building processes as **workflows**. In Step Functions, a workflow defines a business process consisting of a number of **states**. The Step Functions runtime executes the process as a state machine, moving from one workflow state to the next. [21]

The states of a Step Functions workflow are the basic building block for building business processes. A single state in a workflow can either perform a task, define state variables to the next step, branch the execution of the

workflow based on a condition, start executing branches in the workflow in parallel, pause the workflow until a certain time has elapsed or end the workflow with either a success or error status. [51]

For running business logic code, a state in a workflow must be a task calling an external action. This typically means calling another AWS managed service such as triggering the execution of an AWS Lambda function or sending a message to a Simple Queuing Service queue. In the case of a synchronous integration such as a Lambda call, the output of the call is immediately usable by the next state in the workflow. In the case of an asynchronous interaction such as SQS, this is not possible, and to process will only continue once it is again woken up by another component, e.g., the microservice that processed the SQS message. As shown in figure 4.5, persistent ids called **task tokens** can be used to correlate requests and asynchronous responses between a workflow and an external service.

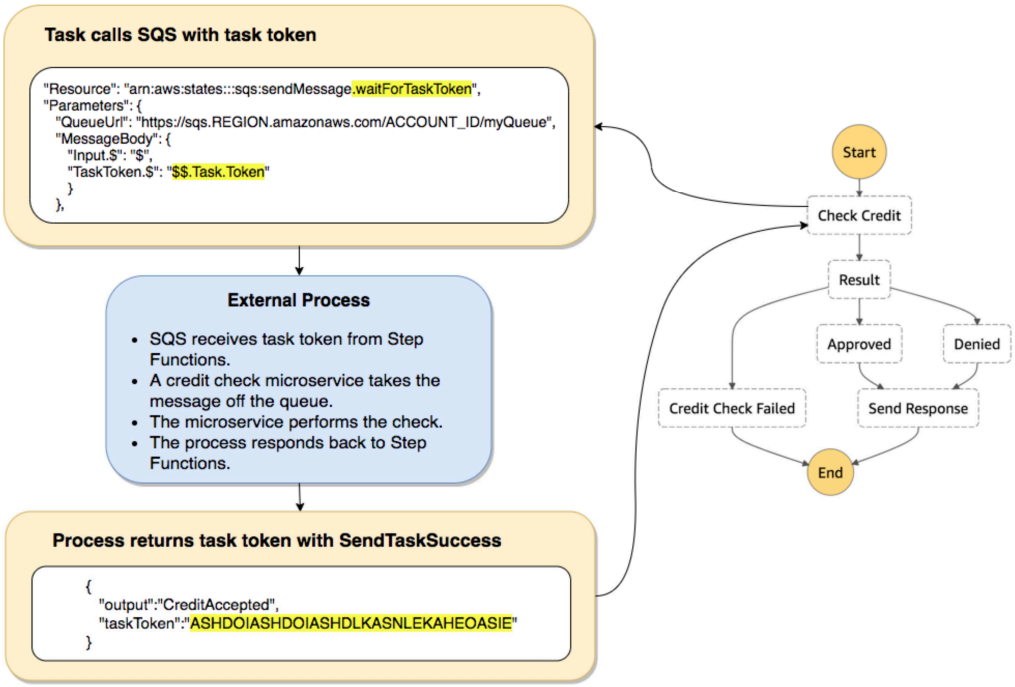

Figure 4.5: An example Step Functions workflow. An external microservice is invoked asynchronously by sending a message to an SQS queue. [51].

In AWS Step Functions, a workflow definition is deployed to the Step Functions runtime. The workflow itself only defines the states of the business process and for running the workflow, one must start an **execution** of the

workflow. An execution is a running instance of the workflow definition. While an execution is running, the Step Functions runtime keeps track of the state of workflow, i.e., what step of the process is being executed and what state variables are defined for this specific execution. [51]

As in a distributed microservice system failures should be considered a norm rather than an exception, it is imperative for business process implementations to cope with failing services. For achieving reliable process execution in a failure-prone environment, Step Functions workflow executions can automatically retry failed steps to overcome intermittent errors [21]. Retry loops can be used to overcome temporary failures that might occur due to e.g., network glitches or failed deployments.

Workflows in Step Functions are defined in a domain-specific language called Amazon States Language. The format for Amazon State Language is Javascript Object Notation (JSON) and workflows are defined in the language as an array of tasks. The AWS console web application offers a visual designer for building Step Functions workflows. The designer parses the JSON-based workflow definition and builds a graph visualization of the steps in the workflow. [51]

AWS Step Functions bears many resemblances to the classical Service Oriented Architecture orchestration approach. In classical SOA, heavyweight workflow tools such as **Business Process Execution Language** engines were used for building business processes [27]. Where the SOAP Web Service -related WS-BPEL process description language was XML-based, in Step Functions JSON state declarations are used for "programming" the process logic. The BPEL engines also offered visual design tools for building workflows, similar to the AWS Step Functions console.

Another resemblance between Step Functions and the classical SOA orchestration approach is the rigid split between the coordinator and the business services. In Step Functions, the workflow is deployed to the Step Functions runtime and it is a component that is always separate and not an integral part of any microservice. The workflow cannot execute any code itself and must always delegate program code execution to another component such as AWS Lambda. This fits very well to the serverless architecture style where the system consists of individual nano-sized cloud functions, but having a separate declarative workflow engine does fit so well to the microservice architecture principle of self-contained stand-alone services.

AWS Step Functions is a valid candidate for building orchestrated processes as described in chapter 3.6.3, but it does not fit very well to the requirements set in chapter 4.1 for a simple cloud computing solution. Any non-trivial workflow requires a number of task steps that call external services, so orchestration workflows would always need to be supported by a fleet

of additional deployments such as AWS Lambda functions. This scatters a Step Functions workflow as a deployment across numerous AWS managed services. Process orchestration is also a responsibility that might often be given to a backend for frontend, but with Step Functions, the orchestrating workflow is always a separate entity from the BFF. To better support the scenarios where orchestration is just one of the responsibilities of the service (e.g., BFF microservice), one needs look at patterns where the process engine can be embedded within the microservice.

4.5.2 Coordinator microservice with Elastic Container Service and Relational Database Service

The microservice architecture pattern strives to enable building complex systems from simple individual components. Following this design principle, instead of using a complex separate runtime for business process execution, one can utilize a lightweight process engine that can be execute processes within a microservice. But to take advantage of running in a cloud runtime, the process engine should not bring any extra unmanaged deployment dependencies (such as vendor-provided virtual machines or persistence solutions) but it should be able integrate with managed AWS services.

For building microservices that need to perform process orchestration, one can take a similar approach as is presented in chapter 4.4.1 for building a flexible backend for frontend solution by utilizing containers and AWS managed services. Like the BFF, a microservice that coordinates other processes should be deployable to Elastic Container Service as a Docker container. And for persisting process state, the coordinator should be able to utilize AWS managed persistence solutions. With this design, one can build complex stateful processes in business microservices or in backends for frontends, but still take advantage of the features offered my managed cloud services.

Examples of embeddable workflow engines include Spring State Machine[8] (by Pivotal), NFlow[9] (by Nitor) and jBPM[10] (by Red Hat). All of these can be embedded in microservices that run on Java virtual machine. The process engines can be taken into use by adding the necessary software libraries to the microservice Docker container and including the process definition in the microservice deployment. The mentioned process engines have some minor differences in the process definition design, for example jBPM uses a specialized domain-specific language where NFlow and Spring State Machine

[8] `https://projects.spring.io/spring-statemachine/`
[9] `https://nflow.io/`
[10] `https://www.jbpm.org/`

use standard Java programming language constructs. Figure 4.6 presents a sample workflow definition in NFlow. As with AWS Step Functions described in chapter 4.5.1, the process engines then run executions of the workflow in a state machine according to the states and transitions defined in the process definition.

```
public class ProcessCreditApplicationWorkflow extends
    WorkflowDefinition<ProcessCreditApplicationWorkflow.State> {

  public enum State implements WorkflowState {
    createCreditApplication(start, "Create new credit application"),
    startCreditDecisionWorkflow(normal, "Start credit decision workflow"),
    waitCreditDecisionWorkflow(normal,
        "Poll for result of credit decision process"),
    createLoan(normal, "Create the loan based on application"),
    transferMoney(normal, "Transfer money to deposit account"),
    transferMoneyFailed(normal,
        "Transfering money failed, reverse creating loan"),
    updateCreditApplication(normal, "Update the credit application state"),
    manualProcessing(manual, "Process must be handled manually"),
    done(end, "Credit application has been completed.");
  }

  public ProcessCreditApplicationWorkflow() {
    super("processCreditApplication",
        createCreditApplication, manualProcessing,
        new WorkflowSettings.Builder().setMaxRetries(3).build());
    permit(createCreditApplication, startCreditDecisionWorkflow);
    permit(startCreditDecisionWorkflow, waitCreditDecisionWorkflow);
    permit(waitCreditDecisionWorkflow, createLoan);
    permit(waitCreditDecisionWorkflow, updateCreditApplication);
    permit(createLoan, transferMoney);
    permit(transferMoney, updateCreditApplication, transferMoneyFailed);
    permit(transferMoneyFailed, manualProcessing);
    permit(updateCreditApplication, done);
  }
```

Figure 4.6: A sample NFlow workflow definition[12]. The workflow has a single start state, end state, six normal states and a manual state where human intervention is required to complete the process. The workflow also uses automatic retries for failed operations.

For keeping track of a process executions state and defined variables, the embedded process engines require access to a relational database. In Amazon Web Services, one can use a managed persistence solution called **Relational Database Service** (RDS) to provide this capability. The RDS managed cloud service can run standard relational database software products such as PostgreSQL or MySQL [21], so the process engines can use RDS for storing persistent state without any AWS-specific integrations. As a managed service, the RDS runtime then offers features minimizing the need for manual process data store maintenance such as elastic scaling of storage and

automated monitoring and backups [21].

Embedded lightweight business process engines enable building coordinator microservices that can run stateful complex business processes with support for actions such as automatic retries and manual human tasks. As the process engine can be integrated to any microservice (that fulfills the technical requirements), orchestration capabilities can be added to an existing microservice to support a specific use case. This is especially important in the context of a backend for frontend. As described in chapter 3.6.3, a BFF is a good candidate for an orchestrator, and as the solution of using ECS and an embedded process engine aligns well with the BFF design from chapter 4.4.1, the solution presented in this chapter can be used to add orchestration capabilities also to backends for frontends without the need of a significant refactoring.

4.6 Summary

The previous chapters have presented a reference architecture for building an event-based microservice system in the public cloud. The design enables to use of propagating changes in the system with an event stream built with Amazon Managed Streaming for Apache Kafka, running microservices packages as Docker containers in the Elastic Container Service runtime, building backends for frontends with API Gateway, Application Load Balancer and containers and running orchestrated processes in a lightweight workflow engine storing persistent state in Amazon Relational Database Service.

The reference architecture presents a design for implementing a microservice system using a small subset of the available Amazon Web Services platform features. The design goal for the architecture was to define a model for a simple, understandable cloud computing solution for implementing a microservice system while taking advantage of the cloud platform features. Key feature in the reference architecture is the choice of using a Docker container as the deployment artifact for microservices. In the reference architecture, deploying services as containers provides the flexibility required for running very heterogeneous microservices while preserving the ability to benefit from cloud runtime capabilities such as elastic scaling.

[12]Source code from `https://github.com/NitorCreations/nflow`

Chapter 5

Discussion

Chapter 3 described architectural patterns for building a microservice system from decomposed, independent components. Chapter 4 presented how the patterns can be implemented in Amazon Web Services cloud computing platform. The software architecture structures and implementation guidelines presented in this thesis form one reference architecture for a distributed system designed for cloud computing deployment. In this chapter, the reference architecture is evaluated and its benefits and disadvantages are discussed. The findings presented in the thesis are also compared to the authors empirical experience of building large enterprise systems with microservice architecture.

5.1 Analysis of the proposed architecture

What is apparent from the presented reference architecture, is the inevitable **complexity of distributed computing**.

In a monolithic application, a business process might be implemented by a REST API accepting a command which is executed by the application by synchronously calling a sequence of functions in the same process. These functions would then update the monolithic database in a single transaction preserving the atomicy and consistency of state modifications. The monolithic application might be packaged in a single installation package (such as RPM or deb), released to a package repository and installed on a number of servers by the operating system package manager.

In the case of the microservice system presented in this thesis, the business process would be implemented by a fleet of independent processes each responsible for a single business task. Persistent state in the system would be scattered to a number of data stores that only asynchronously form an

eventually consistent aggregate. In the case of implementing the business process with choreography, the implementation would at runtime manifest in the microservice system in the flow of asynchronous events in multiple topics of a messaging solution. The microservices in the system would publish interfaces for other services to consume, and patterns such as consumer driven contracts would be used to try avoid breaking changes. Deploying the system to Amazon Web Services would consist of one deployment per microservice to a cluster or virtual machines, possibly abstracted away as in the case of AWS Fargate, each running a number of microservices isolated in Docker containers. To put it briefly, the complexity creep of this solution compared to a single monolithic application is outright massive.

However, we must remember the context from chapter 2. The microservice system reference architecture presented in this thesis is not fit for simple, small applications. It is designed for enterprise applications that are in any case bound to grow huge both in the lines of code and in the amount of constantly changing business rules and their never-ending exceptions. A monolithic enterprise application at a certain point of its growth passes a point when adding new features or fixing defects often leads to breaking other parts of the application. The pace of development grinds to a halt when all changes become extremely risky. Developers react to this risk factor by keeping changes as local as possible, which then leads to eroding the application architecture as use case logic is spread across the whole code base [35]. The microservice system architecture presented in this thesis is designed for applications in this problem domain. The reference architecture tries to apply the tried and tested principles of the Unix philosophy to enterprise applications in the cloud to build software "*out of simple parts connected by well-defined interfaces* [34]".

The architecture described in this thesis approaches the problem of coupling from the bottom up, from the data. Integrating different parts of a system via a shared database is an extremely common pattern in enterprise applications [29]. As this is the most common and the most severe cause of coupling, breaking this interconnectedness is at the heart of this reference architecture. Splitting all persistent state to service-private data stores is from where all the other patterns follow. This single choice causes the paradigm shift from hard consistency to eventual consistency. This thesis has described how event-driven architecture can be used to cope with this distributed data storage model and how events are at the key of avoiding tight coupling between microservices.

Event-driven architecture promotes loose coupling between different components of the system. However, the flip side of the coin is that the resulting system is hard to reason about. As the interaction of different services are

decoupled via a message broker, tracing and debugging an interaction is a non-trivial effort. This decoupling also applies in time as discussed in section 3.3.2, which has a number of benefits as discussed in this thesis, but it also adds complexity to the system and makes debugging the interactions even more difficult.

5.2 Bringing microservices to the cloud

Most system architecture patterns now have take cloud computing into account in one way or another, and microservice architecture is no exception. Microservices as an architecture design pattern has properties such as autonomy of components that align fairly well with the characteristics of common cloud computing platforms. However, the architectural style of microservices was "discovered" and defined (by e.g., Fowler and Lewis [15]) over five years ago in 2014 when also the cloud computing landscape was radically different from what it is today. The term microservices predates for example AWS Lambda and the entire Function as a service (FaaS) concept. So while microservice architecture is very often used in cloud deployments and it can quite well support taking advantage of cloud platform features, it is by no means a cloud-native or serverless architecture style.

This mismatch of serverless computing and microservice architecture is quite apparent in this thesis for example in chapter 4.5.1 when analysing different options for implementing microservice orchestration in the cloud (research question four). AWS Step Functions is the best suited Amazon managed service for the task, but is not a very good fit for a microservice system as Step Functions is a solution targeting serverless cloud-native architectures which microservice architecture does not represent. Similarly, in analysing event stream implementations options in Amazon Web Services (research question three), as a single managed service, only Amazon Managed Streaming for Kafka is a good fit for the requirements needed by event-driven microservices and Amazon MSK is only a very recent (2019) addition to the AWS service palette.

The public cloud reference architecture in this thesis is built with the explicit goal of a simple, understandable solution. This design choice leads to the implementation being a "generic solution" by design. The implementation described in the thesis thus does not attempt to optimize the performance or the scalability of the microservice system implementation. But it is also notable that in search of a simple understandable cloud microservice architecture, the reference architecture on purpose does not use the more intricate features of cloud platforms. And one can very well bring up the

argument that the reference architecture does look like more of a traditional in-house data center system architecture than a serverless cloud computing architecture. This can however be seen also as a positive attribute of the reference architecture presented in the thesis.

A system architecture using fairly common open-source components is portable not only from the cloud to an in-house data center, but also between cloud providers. A microservice system that relies heavily on Docker, Apache Kafka and PostgreSQL is fairly simple to port from the Amazon Web Services cloud to Google Cloud Platform or Azure. But a system built on AWS Lambda, DynamoDB and Step Functions, all proprietary Amazon services, would require a fairly significant rewrite if the system would be ported out of the Amazon cloud. One can also speculate that in the the industry there currently might work a lot more engineers with experience building and running systems on common open-source components than with bleeding edge cloud-native solutions.

Ultimately, only time will tell which architectural styles will stand the test of building large long-lived enterprise systems. For example, classical Service Oriented Architecture (which was very much hyped in the early 2000s) is now seen as a failed experiment driven by commercial interests of large middleware vendors. One can see some similar aspects in cloud-native architectures. But it remains to be seen if in the years to come microservice and/or serverless architecture will also be moved to the category of "it seemed like a good idea at the time". Fortunately, software architecture evolution is an iterative process and also architectural styles that fall out of use contribute to the following generations of architecture design. Microservice architecture is very heavily based on concepts that can now in hindsight be seen as the good parts of classical SOA.

5.3 Findings from empirical evidence

The author has four years of industry experience in designing and developing enterprise systems with microservice architecture, mostly in companies that operate in the finance sector. In this chapter, some findings presented in the thesis are compared to experiences from the trenches of enterprise application projects.

The design of an event-driven microservice system presented in Section 3 puts great emphasis in enabling the development of a system where individual microservices are not tightly coupled to each other. This is a crucial aspect of building a large distributed system. Once components start to become tightly coupled, the possibilities for maintaining and evolving the

microservice system erode quickly. But this is also the aspect that is the hardest to put into practice. Software is developed by developers and all developers are shaped by their past experiences. When new companies and new teams to start developing software with microservice architecture, developers naturally try to use solutions that have worked well for them in monolithic applications. This includes techniques such as passing data via a shared database, making synchronous calls between microservices and creating shared software libraries for reusing code between different microservices. And these solutions do work quite well while the microservice system is small, perhaps up to a dozen services. But once the number of services increases, the interconnectedness of the services, both visible as with synchronous calls and hidden as with shared database access, makes it extremely difficult to change and deploy individual microservices. The system of individual services has been replaced by a distributed monolith where changes and deployments always affect the entire system. This results in a system that would have been better off being designed as a monolithic application, as the system has lost all the benefits of a microservice system but retained all its downsides. Developing an event-driven microservice system requires a guiding hand of an architect that is willing to hold on the principles of the system architecture and sees the difference between an unnecessary architectural compromise (of which there are usually many) and a compulsory architectural compromise (of which there a only a few but they are all the more important to spot).

When operating a large microservice system in production, it is crucial to have proper visibility to the system. In monolithic applications, one can very well get by with just aggregating all application logs to a search index. However, in a microservice system, especially in those that make heavy use of events, the capability to follow the execution of an operation across different services and messaging topics is a prerequisite for deploying the system to production. In practice, the cloud reference architecture described in Section 4 requires also building dedicated tooling for tracing of asynchronous interactions between the different Docker containers, Amazon MSK topics and workflow engine executions. The need for greater runtime visibility is often overlooked when developing microservice systems, and this can cause huge delays in fixing issues when the first performance problems occur or a buggy service has propagated faulty data across the network of services in production.

Chapter 6

Conclusions

With the increasing adoption of public cloud platforms for deploying software, the use of distributed system architectures is also becoming more common. Distributed systems excel is some areas, for example in the ease of scaling the system, but they also present challenges with e.g., system maintainability and data consistency that are not present in monolithic applications.

If an enterprise application has reached a point where further development work is effectively impossible (due to technical debt or new features constantly breaking existing functionality), being forced to rewrite the entire application is a huge and expensive effort. Microservice architecture attempts to avoid this scenario by building the system from autonomous components. But there is an apparent risk of creating a system that collects technical debt faster than a monolithic application due to the complexity of distributed system design.

This thesis has described how event-driven architecture can be used to prevent coupling in distributed applications and presented how to apply patterns such as service choreography and orchestration to a microservice system. Using these patterns helps building microservice systems that fulfill the complex requirements of a long-lived large enterprise application.

When building distributed systems on public cloud platforms, development teams and architects can become overwhelmed by sheer number of possible cloud services and the amount of design decision one has to make when first building a cloud architecture for a project. This thesis has defined a reference architecture that can be used as a starting point when designing a cloud-based system architecture for a microservice system. From the fairly simple reference architecture, development teams can then start evolving the system architecture to suit the specific needs of application being developed.

6.1 Further work

Building distributed systems with microservices and deploying these system to public cloud platforms are vast topics. The focus of the thesis is in architectural design, and many of the non-functional aspects of microservice systems and cloud deployments are merely touched upon.

Building large software applications is never only a technical challenge but it is always also an organizational one. One of the promises of microservice architecture is that cross-functional teams should be able to build and deploy microservices autonomously from other teams. This in turn helps in scaling the development effort of a large software system. Further research in this topic could help identify software design patterns for microservice that help or hinder this organizational scaling effort.

The thesis also leaves open the financial aspects of the presented public cloud reference architecture for microservice system. Managed services in public cloud platforms have very different pricing models, and the choice of a microservice runtime or data storage solution can have a significant impact on the cost-effectiveness of running a microservice system in the cloud. A price study comparing the costs for running microservice with different cloud platform managed services would help architects in choosing architectural patterns also with cost-effectiveness in mind.

Bibliography

[1] Bailis, P., and Ghodsi, A. Eventual consistency today: Limitations, extensions, and beyond. *Queue 11*, 3 (2013), 20.

[2] Balalaie, A., Heydarnoori, A., and Jamshidi, P. Migrating to cloud-native architectures using microservices: an experience report. In *European Conference on Service-Oriented and Cloud Computing* (2015), Springer, pp. 201–215.

[3] Banavar, G., Chandra, T., Strom, R., and Sturman, D. A case for message oriented middleware. In *International Symposium on Distributed Computing* (1999), Springer, pp. 1–17.

[4] Bonér, J. *Reactive Microservices Architecture - Design Principles for Distributed Systems*. O'Reilly Media, 2016.

[5] Bonér, J. *Reactive Microsystems, The Evolution of Microservices at Scale*. O'Reilly Media, 2017.

[6] Bradley, J., Sakimura, N., and Jones, M. B. JSON web token (JWT)–RFC 7519, 2015.

[7] Brown, K., and Woolf, B. Implementation patterns for microservices architectures. In *Proceedings of the 23rd Conference on Pattern Languages of Programs* (2016), The Hillside Group, p. 7.

[8] Burns, B. *Designing Distributed Systems – Patterns and paradigms for scalable, reliable services*. O'Reilly Media, 2018.

[9] Cerny, T., Donahoo, M. J., and Trnka, M. Contextual understanding of microservice architecture: current and future directions. *ACM SIGAPP Applied Computing Review 17*, 4 (2018), 29–45.

[10] Combe, T., Martin, A., and Di Pietro, R. To Docker or Not to Docker: A Security Perspective. *IEEE Cloud Computing 3*, 5 (2016), 54–62.

[11] Cugola, G., Di Nitto, E., and Fuggetta, A. Exploiting an event-based infrastructure to develop complex distributed systems. In *Proceedings of the 20th international conference on Software engineering* (1998), IEEE, pp. 261–270.

[12] Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., and Safina, L. *Microservices: Yesterday, Today, and Tomorrow*. Springer International Publishing, Cham, 2017, pp. 195–216.

[13] Fowler, M. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[14] Fowler, M. Tolerant reader. `https://martinfowler.com/bliki/TolerantReader.html`, 2011.

[15] Fowler, M., and Lewis, J. Microservices. `https://martinfowler.com/articles/microservices.html`, 2014.

[16] Garcia-Molina, H., and Salem, K. *Sagas*, vol. 16. ACM, 1987.

[17] Helland, P. Data on the outside versus data on the inside. In *CIDR* (2005), pp. 144–153.

[18] Helland, P. Life beyond distributed transactions. *Communications of the ACM 60*, 2 (2017), 46–54.

[19] Helland, P., and Campbell, D. Building on quicksand. *arXiv preprint arXiv:0909.1788* (2009).

[20] Jaramillo, D., Nguyen, D. V., and Smart, R. Leveraging microservices architecture by using docker technology. In *SoutheastCon 2016* (2016), IEEE, pp. 1–5.

[21] Jung, M., Möllering, S., Müller, C., Dalbhanjan, P., Chapman, P., and Kassen, C. Microservices on AWS. `https://d1.awsstatic.com/whitepapers/microservices-on-aws.pdf`, 2019. Amazon Web Services, Inc., New York, NY, USA, Tech. Rep.

[22] Kratzke, N., and Quint, P.-C. Understanding cloud-native applications after 10 years of cloud computing-a systematic mapping study. *Journal of Systems and Software 126* (2017), 1–16.

[23] Lamport, L., Shostak, R., and Pease, M. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS) 4*, 3 (1982), 382–401.

[24] Łaskawiec, S. The evolution of Java based software architectures. *J. Cloud Comput. Res 2*, 1 (2016), 1–17.

[25] Mell, P., and Grance, T. The nist definition of cloud computing (draft) recommendations of the national institute of standards and technology. *Nist Special Publication 145* (01 2011), 1–7.

[26] Michelson, B. M. Event-driven architecture overview. *Patricia Seybold Group 2*, 12 (2006), 10–1571.

[27] Morgan, B., and Pereira, P. A. *Microservice in Action*. Manning Publications Co., New York, USA, 2018.

[28] Nadareishvili, I., Mitra, R., McLarty, M., and Amundsen, M. *Microservice Architecture, Aligning Principles, Practices, and Culture*. O'Reilly Media, 2016.

[29] Newman, S. *Building Microservices – Designing Fine-grained Systems*. O'Reilly Media, 2015.

[30] Overeem, M., Spoor, M., and Jansen, S. The dark side of event sourcing: Managing data conversion. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2017), IEEE, pp. 193–204.

[31] Pautasso, C., Zimmermann, O., Amundsen, M., Lewis, J., and Josuttis, N. Microservices in practice, part 1: Reality check and service design. *IEEE Software 34*, 1 (2017), 91–98.

[32] Pautasso, C., Zimmermann, O., Amundsen, M., Lewis, J., and Josuttis, N. Microservices in practice, part 2: Service integration and sustainability. *IEEE Software 34*, 02 (mar 2017), 97–104.

[33] Postel, J. DoD Standard–Transmission Control Protocol–RFC 761. *University of Southern California* (1980).

[34] Raymond, E. S. *The art of Unix programming*. Addison-Wesley Professional, 2003.

[35] Richardson, C. *Microservice Patterns*. Manning Publications Co., New York, USA, 2018.

[36] Robinson, I. Consumer-driven contracts: A service evolution pattern. `https://martinfowler.com/articles/consumerDrivenContracts.html`, 2006.

[37] Rodger, R. *The Tao of Microservices*. Manning Publications Co., New York, USA, 2018.

[38] Rotem-Gal-Oz, A. Fallacies of distributed computing explained. `http://www.rgoarchitects.com/Files/fallacies.pdf`, 2006.

[39] Salus, P. H. *A Quarter-Century of Unix*. Addison-Wesley, 1994.

[40] Savchenko, D. I., Radchenko, G. I., and Taipale, O. Microservices validation: Mjolnirr platform case study. In *2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)* (2015), IEEE, pp. 235–240.

[41] Specification, CAE. *Distributed Transaction Processing: the XA Specification*. X/Open, 1991.

[42] Tai, S., Mikalsen, T. A., and Rouvellou, I. Using message-oriented middleware for reliable web services messaging. In *International Workshop on Web Services, E-Business, and the Semantic Web* (2003), Springer, pp. 89–104.

[43] Tanenbaum, A. S., and Renesse, R. v. A critique of the remote procedure call paradigm. `http://dare.ubvu.vu.nl/bitstream/handle/1871/2592/11014.pdf`, 1988.

[44] Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., Casallas, R., and Gil, S. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *2015 10th Computing Colombian Conference (10CCC)* (2015), IEEE, pp. 583–590.

[45] Wang, G., Koshy, J., Subramanian, S., Paramasivam, K., Zadeh, M., Narkhede, N., Rao, J., Kreps, J., and Stein, J. Building a replicated logging system with Apache Kafka. *Proceedings of the VLDB Endowment 8*, 12 (2015), 1654–1655.

[46] Webber, J., Parastatidis, S., and Robinson, I. *Rest in practice*. O'Reilly Media, 2010.

[47] Amazon Elastic Container Service Developer Guide.
`http://docs.aws.amazon.com/AmazonECS/latest/developerguide/ecs-dg.pdf`,
2019. Amazon Web Services, Inc., New York, NY, USA.

[48] Amazon Kinesis Data Streams Developer Guide.
`https://docs.aws.amazon.com/streams/latest/dev/kinesis-dg.pdf`,
2019. Amazon Web Services, Inc., New York, NY, USA.

[49] Amazon Managed Streaming for Apache Kafka Developer Guide.
`https://docs.aws.amazon.com/msk/latest/developerguide/MSKDevGuide.pdf`,
2019. Amazon Web Services, Inc., New York, NY, USA.

[50] Amazon API Gateway Developer Guide.
`https://docs.aws.amazon.com/apigateway/latest/developerguide/apigateway-dg.pdf`,
2019. Amazon Web Services, Inc., New York, NY, USA.

[51] AWS Step Functions Developer Guide.
`https://docs.aws.amazon.com/step-functions/latest/dg/step-functions-dg.pdf`,
2019. Amazon Web Services, Inc., New York, NY, USA.

[52] Apache Kafka Documentation.
`https://kafka.apache.org/documentation/`, 2019.